Shadow Computations using Robust Epsilon Visibility

Florent Duguet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Shadow Computations using Robust Epsilon Visibility

Florent Duguet

N° 5167

April 2004

THÈME 3

ISSN 0249-6399

Shadow Computations using Robust Epsilon Visibility

Florent Duguet*

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projets TSI and REVES

Rapport de recherche n° 5167— April 2004 — 109 pages

Abstract: Analytic visibility algorithms, for example methods which compute a subdivided mesh to represent shadows, are notoriously unrobust and hard to use in practice. We present a new method based on a generalized definition of extremal stabbing lines, which are the extremities of shadow boundaries. We treat scenes containing multiple edges or vertices in degenerate configurations, (e.g., collinear or coplanar). We introduce a robust ϵ method to determine whether each generalized extremal stabbing line is blocked, or is touched by these scene elements, and thus added to the line's generators. We develop robust blocker predicates for polygons which are smaller than ϵ. For larger values, small shadow features merge and eventually disappear. We can thus robustly connect generalized extremal stabbing lines in degenerate scenes to form shadow boundaries. We show that our approach is consistent, and that shadow boundary connectivity is preserved when features merge. We have implemented our algorithm, and show that we can robustly compute analytic shadow boundaries to the precision of our chosen ϵ threshold for non-trivial models, containing numerous degeneracies.

Key-words: Computer Graphics, Computational Geometry, Visibility, Lighting, Shadows

* INRIA Sophia-Antipolis and Ecole Nationale Supérieure des Télécommunications,Paris

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Calculs d'ombres en utilisant l' ϵ-visibilité robuste

Résumé : Les algorithmes de visibilité analytique, par exemple les méthodes calculant une subdivision de maillage pour représenter des ombres, sont notoirement fragile, et difficiles à utiliser en pratique. Nous présentons une nouvelle méthode basée sur une définition généralisée des droites poignardantes extrêmes, supportées par les extrémités des ombres. Nous traitons des scènes contenant de multiples arêtes ou sommets en configurations dégénérées (par exemple colinéaires ou coplanaires). Nous introduisons une méthode robuste pour déterminer si chacune des droites poignardante extrême est bloquée, ou est touchée par les éléments de la scène, et ainsi ajoutés à l'ensemble des générateurs de la droite. Nous développons des prédicats de blocage robustes pour les polygones plus petits qu'ϵ. Pour des valeurs d'ϵ plus grandes, les détails des ombres fusionnent et finissent par disparaître. Nous pouvons donc connecter de façon robuste des droites poignardantes extrêmes généralisées dabs des scènes dégénérées pour former des des contours d'ombres. Nous prouvons que notrtre approche est cohérente et que la continuité des contours d'ombres est préservée quand les détails fusionnent. Nous avons implémenté notre algorithme et nous montrons que nous pouvons calculer de façon robuste les contours d'ombres à la précision de la valeur d'ϵ choisie pour des modèles non triviaux, contenant de nombreuses dégénérescences.

Mots-clés : géométrie algorithmique, graphique 3D, visibilité, éclairage, ombres

Contents

Acknowledgements

This work has been done at Ecole Nationale Supérieure des Télécommunications in Paris under supervision of Francis Schmitt. It is however the translation of the french masters thesis internship performed at INRIA Sophia-Antipolis under supervision of George Drettakis, to which have been added implementation details and other contributions.

This thesis, written for the obtention of a "Brique Projet" is the compilation of all the algorithmic and implementation techniques, and explanations on the theoretical framework of **Robust Epsilon Visibility** [DD02].

Chapter 1

Introduction

In this thesis we present several algorithms and techniques related to visibility, shadow computations and computational geometry. The main focus of our work has been to compute shadows for polyhedral scenes using robust epsilon visibility.

The spectrum of shadow algorithms in the literature is wide. From the most hardware oriented to analytic techniques, each has its benefits and drawbacks. We develop here techniques for analytic visibility computations, and shadow computations as an application.

In this work, we focus on robustness issues and problems related to precision. Indeed, computers are not capable of infinite precision computations which are needed for stable and consistent geometric algorithms. Our choice in front of this problem has been to handle degeneracies, and to consider things degenerate when the computer calculations could not be trusted. This choice resulted in a new framework on analytic visibility we called epsilon visibility.

Each computation we do each result and we expect and return is to be considered correct up to an epsilon error threshold. Beyond this epsilon value, results are not fair and even topology consistency is not guaranteed. In some applications, this could be a severe problem, but for shadow computations, a well chosen epsilon leads to fair results with robust algorithms.

This thesis can be seen as an extension, both in terms of techniques, and presentation of the 2002 Siggraph paper Robust Epsilon Visibility, by Florent Duguet and George Drettakis.

Chapter 2

Previous Work

In this chapter, we briefly present previous work in terms of visibility and shadows, with a very short presentation of interval techniques in computer graphics.

The section is structured as follows:

- we first discuss visibility, in particular analytic, approximate visibility and occlusion culling. This part deals with visibility only.
- we next discuss shadows, subdivided in sharp shadows, soft shadows using discrete techniques, and soft shadows using analytic techniques.
- finally a small part on interval techniques in computer graphics.

2.1 Visibility

Visibility is a central problem in computer science. It has been addressed extensively in the literature, in different ways, for specific applications. An extensive study of visibility problems and previous work has been done by Durand in his PhD [Dur99]. We described here three kind of visibility problems : analytic visibility, approximative (discrete) visibility, and occlusion culling.

Analytic Visibility

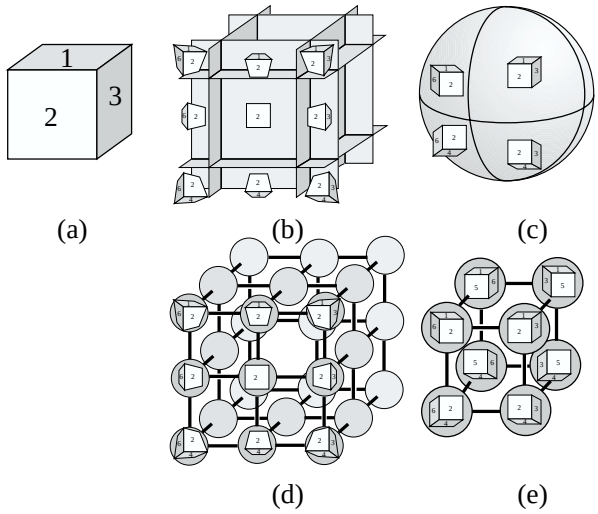
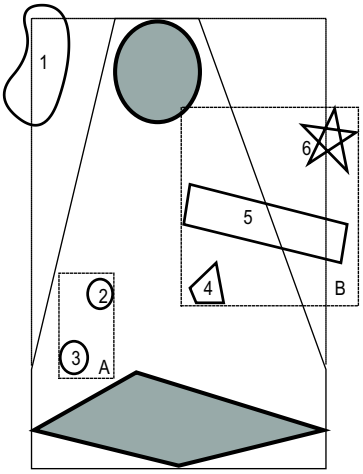


Figure 2.1: Illustration of the aspect graph : (a) Object, (b) partitioning, (c) orthographic projection partitioning, (d) aspect graph, (e) aspect graph for orthographic projection - figure from [Dur99]

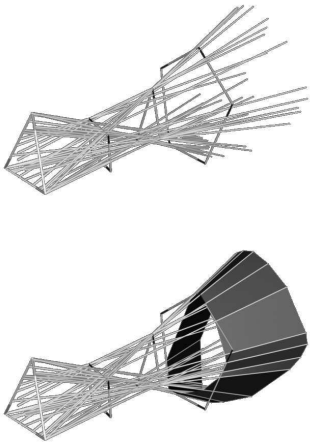
Aspect Graph Model oriented pattern recognition needs a *viewpoint* oriented representation of objects; that is a structure which can code all the possible views of an object. Koenderink and Van Doorn [KvD79] have developed the *visual potential* of an object, known as the *aspect graph*. This technique consists in partitioning viewpoint space into cells. From any two viewpoints of a given cell, the object looks the same in a qualitative point of view (see figure 2.1). We call this invariant the *aspect*. The set of cells is then represented by a graph structure as follows : each node is an aspect of the object; and each arc is associated to a visibility event, that is a transition between two aspects.

Shaft Culling Visibility between surfaces is the most expensive part of radiosity computations. One approach is to compute visibility using a discrete technique such as ray casting methods. Haines and Wallace [Hai93] presented an algorithm which takes advantage of object space coherence. The method is based on the use of shafts which overlap the volume between the emitter and the receiver of an energy transfer. Then a candidate list of elements partially or fully within this volume is generated. This list is thus used for visibility test (ray casting), avoiding unnecessary computations with irrelevant elements. This shaft technique is widely used in lighting simulation algorithms and hierarchy-oriented applications.



Shaft illustration - from [Hai93]


Portals illustration - from [TH94]


Anti-penumbra - from [Tel92a]

Occlusion Culling - Portals and Anti-penumbra Teller and Hanrahan [TH94] proposed an algorithm to compute visibility for architectural scenes, that is indoor buildings. The idea is to first partition the scene into convex polyhedral cells. Within these cells, the visibility computations are trivial. Once the cells are computed, they are linked by portals. For example, a building with rectangular rooms has rectangular cells (the rooms themselves), and the doors define the portals. In order to compute visibility, we compute the set of portals between two cells, and we clip the set of lines between these cells by the portals. We thus obtain a visibility shaft. Each line within this visibility shaft is free. We can compute the partial/complete/non - visibility of any pair of polygons with this structure.
In order to compute the visibility shaft, through a sequence of portals, it is necessary to compute the anti-penumbra and anti-umbra. Teller [Tel92a] proposed an algorithm to build the shaft. To achieve these computations, Teller uses extremal stabbing lines and critical line sets. He gives fundamental definitions later used in analytic visibility.

The Visibility Complex The Visibility Complex is a formal approach of 2D visibility problems addressed by Pocchiola and Vegter [PV96]. This technique is the study of maximal free segments, which are segments of the 2D space, in intersection with no object and with extremities on the limit of objects. See fig 2.2 for an illustration. The authors give an optimal output construction algorithm for such a structure, for smooth objects. Riviere [Riv95], [Riv97] proposed an algorithm for polygonal scenes.

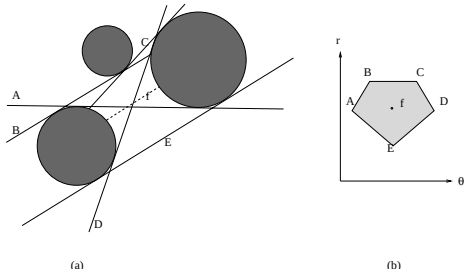


Figure 2.2: Illustration of the 2D visibility complex

3D Visibility Complex The 3D Visibility Complex is an extension of the visibility complex to 3D space. Durand et al [DDP96] extended the visibility complex to 3D scenes for smooth and polygonal objects. The set of maximal free segments of a 3D scene is a super set of a dimension 4 variety (because of the possible multiplicity of segments on a line). Faces of the complex are bounded by tangent segments (critical set of dimension 3), bitangent segments (dimension 2), tritangent segments (dimension 1), and quadritangents, which are the vertices of the complex. It is difficult to describe such a structure since it lies in a 4D space which is hard to draw. For further details on the 3D visibility complex and a clever way to explain it, see [Dur99] and [DDP02]. See fig 2.3 for illustration.

The Visibility Skeleton In lighting simulation algorithms such as radiosity, some visibility information is useful and can be retrieved from the visibility complex. Durand et al [DDP97] presented a data structure storing this data for polygonal scenes : the Visibility Skeleton. The visibility skeleton can be seen as a graph, with nodes being extremal stabbing lines (vertices of the complex, i.e. quadritangents), and arcs being critical line sets of dimension one. Such elements are visibility events of dimension one for nodes and two for arcs. The authors provided a construction algorithm based on a catalogue of extremal stabbing lines, giving adjacencies for the nodes. The visibility skeleton has been used in a global illumination algorithm based on radiosity for which analytic extended source - point form factor computations were achieved. Umbra and penumbra limits are deduces from this data structure. Such visibility computations are analytic and thus exact up to the machine precision.

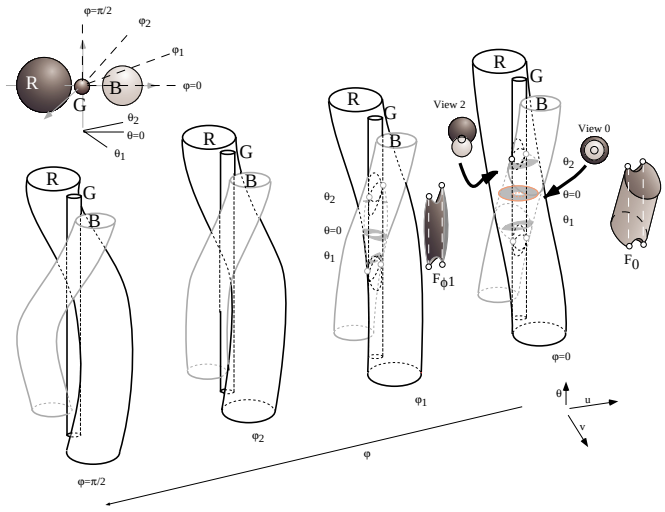


Figure 2.3: Illustration of the 3D visibility complex. A ϕ-slice. Image taken from - [Dur99]

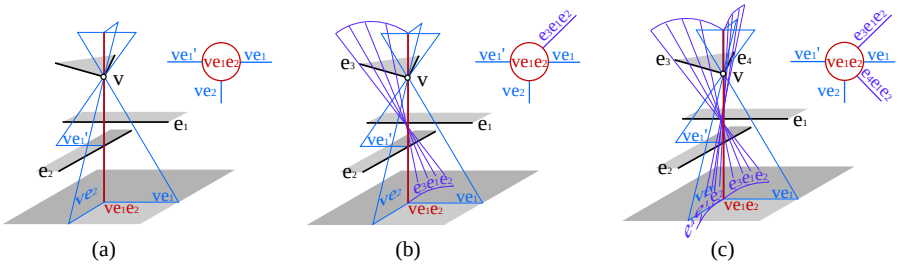
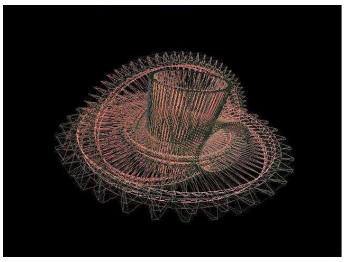


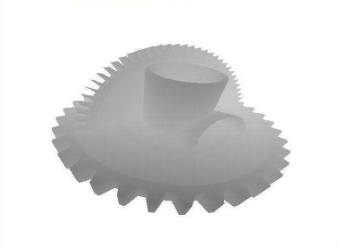
Figure 2.4: Illustration of a visibility skeleton node and arc adjacencies. Image taken from - [DDP97]

Approximate Visibility

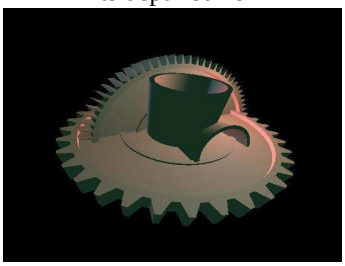
Depth Buffer Hidden surface removal has been an active research topic for several years. In order to display a scene on an image, we have to project the *visible part* of the objects of the scene on the screen. A first algorithm has been presented to draw such objects: the painters algorithm. This algorithm consists in drawing objects on the screen in order of decreasing depth (distance from the screen). Such an algorithm fails for overlapping elements or in the case of unorderable elements (imagine a mikado game with three sticks, each being above another and below the last one). Several algorithms which cut polygonal scene objects have been presented, but the problem remains for curved objects.
Catmul [Cat74] proposed an algorithm which kept in memory a map of distance of pixels drawn, so that a further point did not erase a nearer point and vice versa. This map is known as the depth - or Z buffer and is widely used today. This technique is implemented in OpenGL and available on most graphics cards nowadays. Other techniques are inspired from this one, such as the shadow map (described below).



Wire-frame drawn object



Its depth buffer



Result

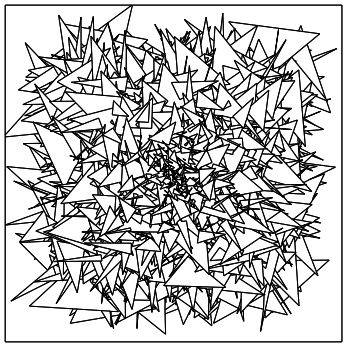


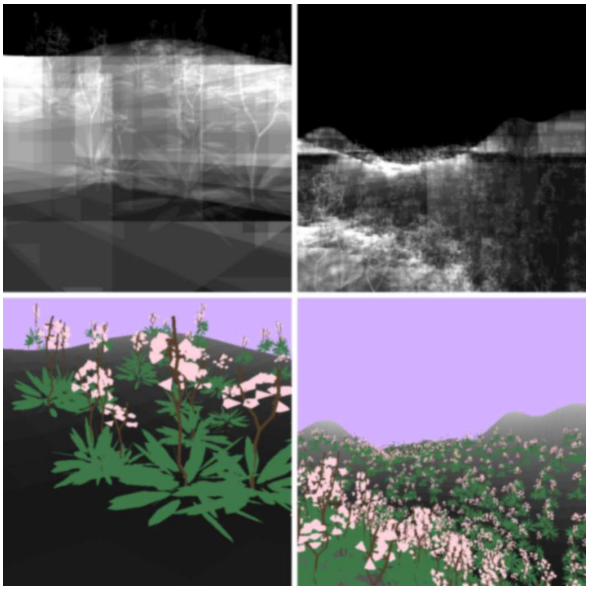
image taken from [SK98]

Approximative Visibility Map Stewart and Karkanis [SK98] proposed an hybrid visibility technique, between analytic and sampled visibility. This method computes the approximate visibility from a point, which is the part of the scene visible from the given point. The algorithm first renders the scene in a buffer in the same way as the depth buffer, giving each pixel a colour corresponding to a polygon it represents. Once this buffer is computed, a graph is extracted from this buffer via a re-computation of vertices positions towards an exact value. The nodes of this graph are the visible intrinsic or apparent vertices, and the arcs are the part of edges visible from the viewpoint. This technique uses the hardware acceleration for expensive visibility computations and for example gives a fair approximation of a form factor.

Occlusion Culling

Walkthroughs are typical applications using large virtual environments. In this application, the user is located in the virtual world as an observer, and does not see the whole scene. Most of the scene elements are hidden by near objects. To optimize rendering, a set of objects is computed, which corresponds to a super set of actualy visible objects. This set is know as the *Potential Visible Set*, it is often computed on the fly with clustering precomputations, depending on the position of the viewer.

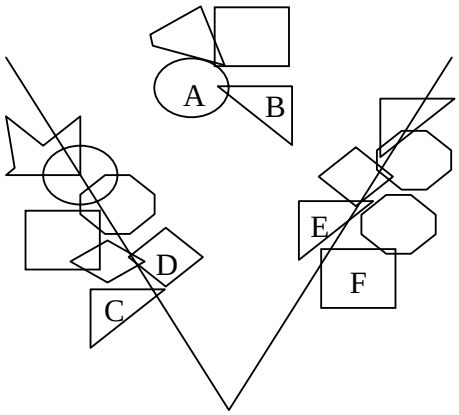
Architectural Geometry Scenes Teller et al [TS91] proposed a technique for hidden object removal for architectural geometry scenes. The technique uses the problem characteristics to transform it into a 2D problem.The input scenes represent buildings with axis aligned walls. The key idea is to subdivide the scene into cells, and to place portals between neighbour cells. Then, two cells may *see* each-other if a line of sight through portals exists. Visibility is computed progressively from a cell through a stab tree. This tree gives the set of visible cells from a given cell. For visibility from a viewpoint, the stab tree has to be searched with lines through the visibility cone. This approach has been extended to less restrictive scene models [Tel92b].



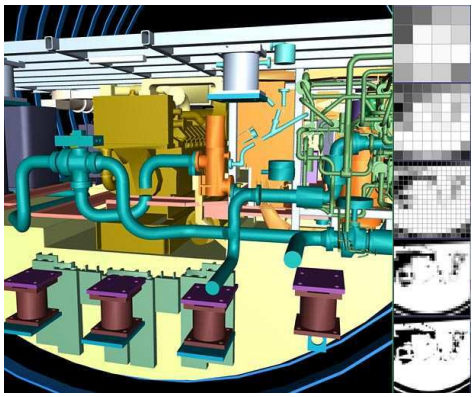
Hierarchical Z-Buffer illustration - from [GKM93]

Hierarchical Z-Buffer Green et al [GKM93] presented a hidden surface removal technique based on Catmull's depth-buffer [Cat74]. This technique gives a hierarchical algorithm of the problem. Instead of using a fixed resolution depth buffer for each object of the scene, the idea is to use a hierarchy of depth buffers, and to test objects with their bounding boxes. This technique has several benefits since it takes advantage of spatial, object-space and time coherence. For spatial coherence: near objects are grouped in hierarchies of bounding boxes. Time and object space coherence are used to define and update a list of visible objects.

Hierarchical Occlusion Maps Zhang et al [ZMHI97] proposed a conservative hidden surface removal algorithm. This technique removes occluded objects at different levels of a hierarchy. Occlusion maps are built from preselected occluders using hardware acceleration: the occluders are rendered, and a conservative algorithm is used to build the hierarchy of occluders. Occluder selection follow a strict list of criteria upon distance, size, shape... Once established, maps are applied on the hierarchy of the scene to solve visibility. The occluder list is updated during the walk-through.



Occluder selection - from [ZMHI97]



Submarine image - from [ZMHI97]

2.2 Shadows

The literature on shadow algorithms is vast, and we do not intend to address it exhaustively. Woo et al [WPF90] presented a survey on shadow algorithms, which is an excellent reference for earlier shadow algorithms.

Sharp Shadows

Shadow Map Williams [Wil78] proposed an algorithm to compute sharp shadows from directional sources, from curved objects to curved surfaces. This technique is based on the depth buffer technique: the scene is rendered from the viewpoint of the source (point or directional), and the depth buffer is stored as the shadow map. We associate the transformation matrix to the map. Then, we render the scene from the viewpoint of the camera. Finally, for each pixel of the resulting image, we restore its 3D coordinates, and compute its depth with respect to the viewpoint of the source. The resulting depth is then compared to the one given in the shadow map, and if above the stored value, the point is in the shadow. SGI Origin 2000 and nVidia GeForce 3 cards implement this algorithm in hardware. The main benefit of this technique is that every element which can be rendered with the ZBuffer can thus be shaded. However, precision problem appear, and resolution of the shadow map quickly become critical.

Perspective Shadow Maps Stamminger and Drettakis [SD02] improved the shadow map technique to overcome the main shadow map drawback: aliasing. The shadow map technique is based on a global shadow map for the whole scene whatever the viewer position is, the resolution of the map is thus insufficient when the scene is closely examined. For large scenes, the problem quickly result in imprecise and aliased shadows. The idea of the method is to adapt the shadow map to the current viewpoint, by computing the map after perspective projection.

standard shadow map
courtesy of M. Stamminger

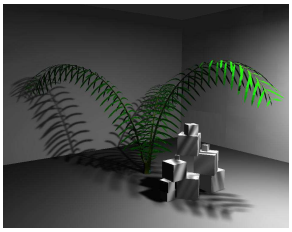
perspective shadow map
courtesy of M. Stamminger

Illustration of shadow volumes from [EK]

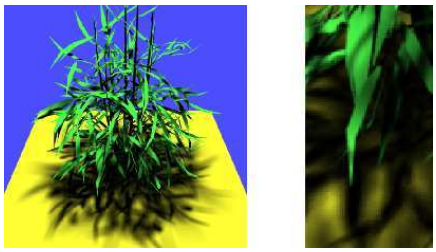
Shadow Volumes Crow [Cro77] introduced the shadow volumes. lgorithmfor a given point or directional light source, polygonal elements defined by the source and the scene edges are cast into the scene to define shadow boundaries. The polygonal elements are given by the source position/direction and the silhouette of objects. This technique is used for interactive display of sharp shadows. The computations of shadows can be performed using the stencil buffer, avoiding numerical failures, and most recently, Everitt et al [EK] gave a robust version of this technique.

Soft Shadows

Soft Shadow Textures Soler et al [SS98] proposed a technique to compute soft shadows based on textures. For a given source-receiver pair, each in-between blocker is approximated by a flat blocker parallel to the source or the receiver, at a given distance. With this given set of flat blockers, a convolution of the source and the blocker images is done, giving a shadow image. The resulting image is applied as a texture to the receiver.



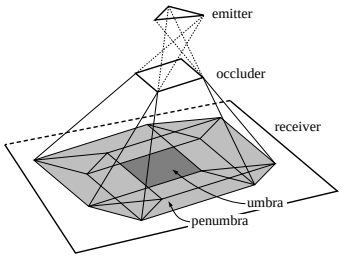
example of soft shadow textures from [SS98]



images from [ARHM00]

Image-Based Methods Agrawala et al [ARHM00] presented a couple of techniques for rendering soft shadows. The first one produces soft shadows at an interactive rate, the second produces high quality images including soft shadows. The first is Layered Attenuation Maps, is based on the use of Layered Depth Images [SGHS98], built from sampling points on the light source. The maps are computed from the LDI and modulate the illumination of the image. The second technique is a hierarchical ray tracing technique which is achieved through the shadow maps instead of scene geometry. The source is sampled at uncorrelated positions, avoiding artifacts from the previous method.

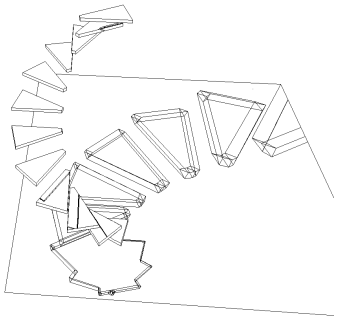
Discontinuity Meshing Heckbert [Hec92] and Lischniski et al [LTG92] studied the discontinuities of the radiance function due to the presence of objects between light sources and receivers; for example in a typical radiosity light transfer. These discontinuities are of several kinds and orders. Order 0 discontinuities are generated by contact or intersection of objects. Order 1 and 2 are generated by occluders between source and receiver. These discontinuities appear along visual events, which are changes in the visibility of the source from the receiver. Algorithms related to visual events were already presented by Teller in [Tel92a]. These discontinuities are projected on the receiver and a constrained triangulation is thus computed, so that no visibility event is present on subdivided polygons. This technique has been used in a radiosity algorithm addressing one of its main drawback.



discontinuities in radiance function
from Heckbert [Hec92]

image using improved radiosity algorithm
from Lischinski et al [LTG92]

Backprojection Drettakis and Fiume [DF94] and Stewart and Ghali [SG94] presented techniques to accurately compute lighting from extended sources. These techniques are based on the concept of back projection. A backprojection instance at a point P, with respect to a source is the set of polygons forming the visible parts of the source at P. The backprojection in a region is a data-structure containing all bp-elements (intrinsic and apparent vertices), such that from any point P in the region, these elements projected onto the source define the backprojection instance. The backprojection is constant in each full discontinuity mesh cell. Then, the irradiance from a constant area source can be computed analytically for each point in such region, with this information. Once the discontinuity mesh and the backprojection are computed, several images can be computed with little additional computational expense.



discontinuity meshing
from [SG94]

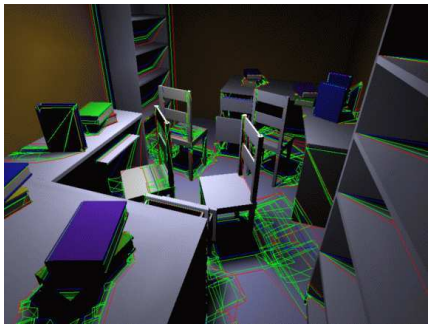


image using backprojection, and superimposed
discontinuity meshing
from [DF94]

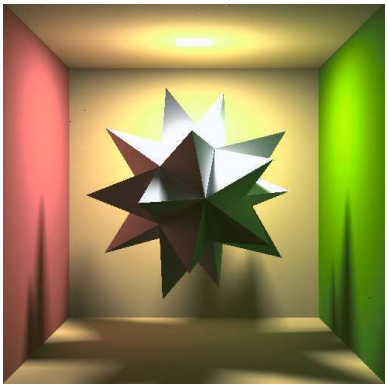


image using vertex tracing
from [SR00]

Vertex Tracing One of the main drawback in radiosity is the lack of precision in shadow boundaries. A *final gather* step has been introduced to compute accurate soft shadow boundaries for radiosity solutions. In order to achieve this final step, Stark et al [SR00] introduced vertex tracing. The idea of this technique is to compute, for each pixel of the final image, the contribution of each source (each surface in radiosity). This final computation is done with analytic computations using a modified version of the Lambert formula, expressed at vertices. Two kinds of vertices are considered in this formula : intrinsic vertices which are vertices of the initial mesh visible from the point, and apparent vertices which are intersection of edges. Then, each visible part of source define a lighting slice, which contributes to the illumination of the pixel.

2.3 Interval Techniques

Epsilon Geometry One of the main drawbacks of using floating point units (FPU) is the lack of precision of such computations. The IEEE-745 norm on FP computations is indeed satisfactory for range of applications, but the precision obtained is often insufficient for geometrical computations. Salesin et al [SGS89] presented a general framework for geometrical predicates from imprecise computations. Each geometrical element is thus flattened of a user-defined epsilon and predicates are allowed to return a result as *unknown*. The authors presented an extensive study of typical 2D geometrical problems.

Interval Analysis Snyder [Sny92] presents a formal framework on interval analysis, especially for problems arising in computer graphics applications. Two algorithms SOLVE and MINIMISE are presented in an interval arithmetic computational approach. He describes several applications of these algorithms for computer graphics, mainly in the implicit curves / surfaces field. The key idea is to use inclusion functions which are functions with suitable properties over intervals. Problems defined with such functions can be solved by an iterative approach in the spirit of divide and conquer algorithms.

Chapter 3

Framework

As explained in the previous work, analytic visibility is a general geometric framework for several applications including shadow computations, occlusion culling, etc. Elements of interest for such applications are visibility events. We will describe in this chapter tools used for visibility computations, and a general framework on the topic, regardless of precision problems or degenerated configurations; these topics are detailed in following chapters.

This chapter is an introduction to analytic visibility, to the underlying concepts and tools. For an extensive detailed presentation of analytic visibility, the reader should refer to Durand's PhD Thesis [Dur99].

For the rest of the discussion, a **scene** is a set of polyhedra, given in a 3D space. These polyhedra are made of **vertices** (points with given 3D coordinates), **edges** (finite segments between two points), and **faces** (2D surfaces between points and edges, restricted to planar convex polygons).

The visibility complex [DDP96] is a good theoretical framework on visibility, which uses concepts like tangency. These concepts seem obvious in the mathematical field, but need proper definition to understand problems which eventually appear in computer graphics.

In the first section, we will describe the set of predicates we need for visibility computations, then we define the extremal stabbing lines, which are visibility events of dimension 0; then we define swaths, which are visibility events of dimension 1; and finally we give a brief description of the visibility skeleton introduced by Durand et al [DDP97].

3.1 Visibility predicates

As stated, we only consider scenes made of convex planar polygons. More complex geometry is not addressed in this thesis. By the way, we are confident in the assumption claiming that all graphical objects can be approximated by a polyhedron, and thus triangulated to fulfill the previous requirements.

Also, we consider, for the simplicity of the discussion, that we have infinite precision computation tools at our disposal. This simplification assumption will be overridden in the next chapter.

Visibility studies light **rays**, which are considered infinitely thin, as lines in geometry, and directed from an origin to a destination. Thus, a ray has an origin (point in the 3D affine space), and a direction vector (in the 3D underlying vector space). A ray goes along a line which is called its supporting line.

We call **object** a scene element, whether it be a vertex, an edge, a face, or any composition of objects. Note that an edge contains its boundary vertices and is thus a composition element, as for the face containing its boundary edges.

Hit Criterion

The hit criterion defines whether a ray hits an object or not. Every kind of object may be hit by a ray.

A ray **hits** an object if and only if at least a point of the ray lies on the object. Note that in this definition, objects may be defined by single points, edges, faces, opened polyhedrons, etc.

This criterion can be extended to paths, which can be seen as curved rays.

Blocking Criterion

The blocking criterion defines whether a ray is stopped by an object or not. Note that this criterion applies on the neighbourhood of the object. For example, a vertex might be a blocker, even if its spatial extend is null, since it may have neighbouring faces.

Intuitive definition: a ray is **blocked** by an object at P if and only if the ray crosses the surface of the object. That is for example, for a solid shape, if the ray steps inside the object at point P.

Formal definition: Let P be the intersection point of the object and the ray. Let ε be a strictly positive value, let S_ε be a sphere around P of radius ε, let P^- and P_+ be the intersection points of the ray supporting line and S_ε, P^- before P in the ray's direction. A path (continuous set of points) between P^- and P_+ is said to be **free** if and only if there is no point which both lie on the path and on the object except for P_+ or P^-. A ray is **blocked** by an object at point P if and only if all the following conditions are satisfied:

1. The ray hits the object at P.
2. There is an ε for which there is no free path between P_+ and P^-

Tangency Criterion

The tangency criterion is complementary to the blocking criterion. If a ray hits a surface, it is either a blocked or tangent to the object.

Intuitive definition: a ray is **tangent** to an object if it grazes the object.

Formal definition: a ray is **tangent** to an object at point P if it hits the object at point P, and is not blocked by it there.

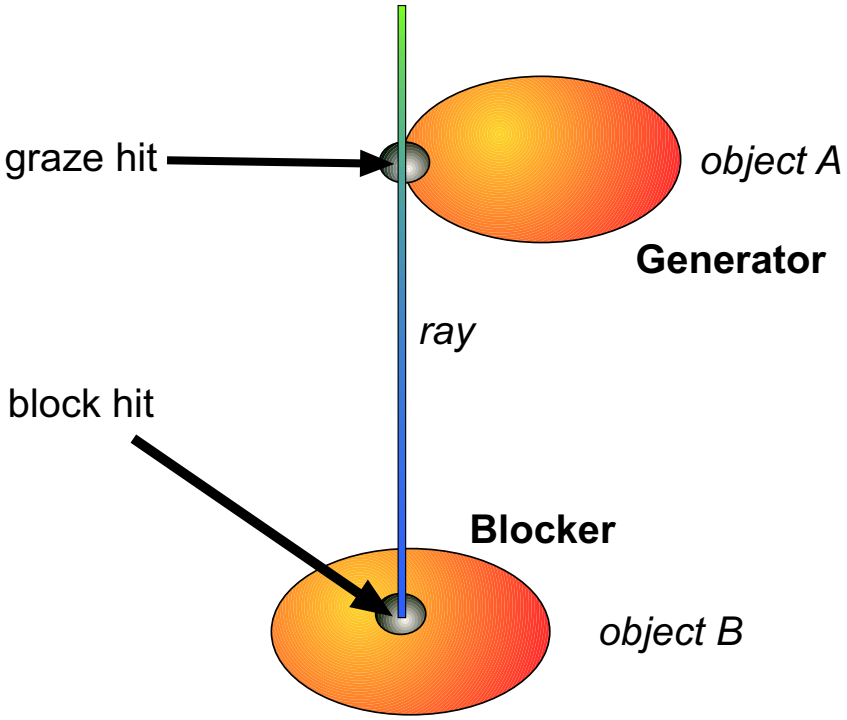


Figure 3.1: Ray classification illustration

Ray Classification

A ray is said to have no interaction with the object if it does not hit the object.

As a direct consequence of the previous definitions, for a given object, an interacting ray is either (at each interaction point), blocked, or tangent. Thus, we now call objects **blockers** with respect to rays if the ray is blocked by an object, or **generators** if the ray is tangent to an object.

3.2 Extremal Stabbing Line

Teller [Tel92a] first introduced the concept of extremal stabbing lines (ESL). The concept has been more formally defined by Durand et al in [DDP96], as quadritangents. A tangency criterion indeed sets one degree of freedom, out of the four available in the line space. Degenerate configurations, with tangency multiplicity greater than four are not addressed in this chapter, by the way, they are the main motivation of our work, and are extensively studied later on.

In [DDP97], Durand et al claimed that for non degenerate configurations, extremal stabbing lines can be classified into a catalogue. The main types of extremal stabbing lines are VV, VEE and E4, other including faces correspond to handling some degenerate configurations with the catalogue tool. Besides, degeneracies are most frequent since lots of objects and structures such as buildings contain degeneracies (aligned vertices, collinear or coplanar edges). By the way, for the clarity of the discussion, we only present here hints for extremal stabbing lines, and study VV,VEE and E4 *generic* ESLs.

Catalogue Approach

An extremal stabbing line is a line defined by generators which constrain its degrees of freedom. Constraining a line to pass through a point (vertex) decreases by two its degrees of freedom, and by one for a line (edge). See figure 3.2 for illustration.

- Two distinct vertices define in a unique manner a line, which is a VV extremal stabbing line.
- A vertex and two edges may define and extremal stabbing line, if the edges, seen from the vertex, appear to intersect on a single point, different from the vertex, which is called apparent vertex. This configuration defines the VEE extremal stabbing line.
- Four edges, supported by four lines may define one or two ESLs. Indeed, if the two lines hitting the four lines actually hit the four edges, the four edges may define two extremal stabbing lines.

Besides this degree of freedom concern, an extremal stabbing line shall also be a maximal free segment.

Algebraic Approach

Before reading this part, please refer to Appendix A for an introduction to Plücker coordinates, and line space.

The generators of the scene (vertices and edges) may be seen as linear maps in the line space (of dimension 6), and the lines through these generators as the real part of the kernel of these linear maps. Indeed, a real line $(\vec{u}, \vec{v})$ runs through a vertex V, if and only if $\vec{V} \times \vec{u} \quad \vec{v} = 0$. This vector equation can be rewritten so that the line appears as an element of the kernel of a linear map. For an edge, the linear map is the Plücker form with the supporting line.

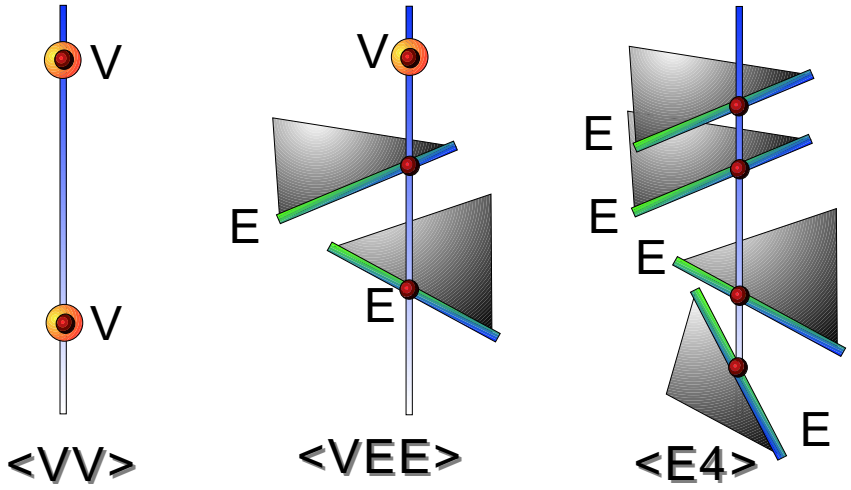


Figure 3.2: Three generic kinds of extremal stabbing lines : VV, VEE and E4

Thus, if the intersection of kernels associated to scene elements contains a finite set of real normalised elements, this set of elements may define an extremal stabbing line. Note that this approach automatically handles degenerate configurations. The algorithm proposed to compute the E4 ESL is inspired from this approach.

For a given set of linear maps, the computation of the dimension of the intersection between the kernels and the normalised variety and real variety is possible and detailed in appendix A. This algorithm is not straightforward and not necessary here, thus not detailed.

Algorithm

We present here algorithms to compute the extremal stabbing lines in the Plücker space (see Appendix A). These algorithms take as input a set of generators, and as output one or two lines generated.

VV

The algorithm to compute a VV ESL is trivial since the line is uniquely defined, we know at least one point on the line, and the line direction is given by the vector $\vec{u} = V_2 \quad V_1$, V_2 and V_1 being the two input vertices.

VEE

The algorithm to compute a VEE ESL is divided into three steps. The first for the direction of the line, the second step for the position of the line (trivial since we know a vertex of the line), and the third step to check if the line passes through the two edges.

Let V be the vertex of the input set of generators, $E_1 = [A_1, B_1]$, and $E_2 = [A_2, B_2]$ be the two edges, see 3.3 for illustration.

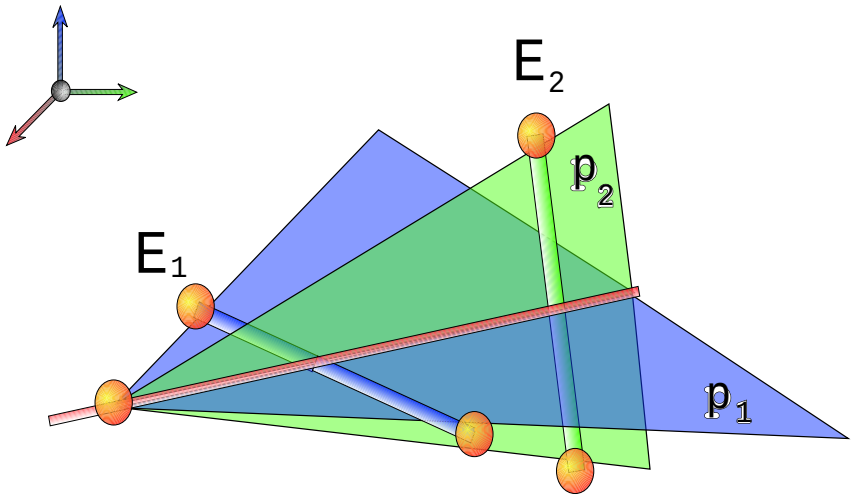


Figure 3.3: VEE ESL computation technique

We first compute the two planes containing an edge and the vertex : $\pi_1 = (A_1, B_1, V)$ and $\pi_2 = (A_2, B_2, V)$, with normals $\vec{n}_1$ and $\vec{n}_2$. Then, the direction of the line is given by the vector $\vec{u} = \vec{n}_1 \times \vec{n}_2$. We know a vertex of the line, given in the set of generators, making the second step trivial.

Finally, to check whether the line actually runs through the two edges, we test the positions of the vertices of the edges with respect to the plane formed by the other edge. That is, the line runs through E_1 if B_1 is on one side of π_2 and A_1 is on the other side.

E4

The algorithm to compute E4 ESLs is a bit more complicated. It is based on the one of Teller [Tel92b]. As for the previous algorithm, we first compute the line, and then check whether the line runs through the four edges.

For a better understanding of the following algorithm, the reader should take a look at appendix A, which deals with Plücker parametrisation of lines, used here.

The four edges supporting lines are expressed in Plücker coordinates (as detailed in appendix A), and a 4x6 matrix is thus computed. We look forward to extract the kernel of this linear map, made of two vectors, in the generic case, which we suppose here.

Teller proposes an algorithm which uses a singular value decomposition, but since all computed elements are not necessary here, we have a different, less expensive approach. We compute a Gauss reduction of the matrix which saves the singular values (operations of the kind : $L_i := L_i + \sum_{j<i} L_j$). We allow columns permutations, if stored, and thus obtain a partially upper triangular matrix. We then rise the pivots to obtain a partially diagonal matrix. We then deduce the kernel elements with the two last columns.

Once the kernel is computed, we have two vectors, say l_1 and l_2. Note that the Plücker space is a projective space, which represents *real* and *imaginary* lines. Given these two vectors, a finite number of lines shall represent real lines, with a unit direction vector. We use an algorithm proposed by Teller in [Tel92b] to compute the resulting line. Several configurations are possible :

- The two lines are real
- One line is real, say l_1 and the other is imaginary
- The two lines are imaginary

The first case is impossible for the following reason : all linear combination of the two real lines is a real line, which would mean that the kernel is of dimension one in the normalised real lines variety. A whole swath would run through (or be coplanar to) the four edges which means that the edges are in a degenerate configuration. This case is not studied here, and we avoid such configurations a priori.

In the second case, all linear combination of the two lines, with a non null weight for l_2 leads to an imaginary line, which is not an expected result here, the second weight of the linear combination is thus null. Our solution is l_1, and is unique (save for its orientation).

In the third case, both weights shall be non null. Since we are in a projective space, we consider a linear combination with a weight set to one (1), and the other noted as λ. we then compute the value of λ so that the resulting linear combination is the parametrisation of a real line. We have to solve a second degree polynomial equation, which leads to one or two solutions (since the discriminant is always positive).

Finally, to check whether the line(s) actually run(s) through the four edges, we do for each edge the following test. We compute the $\vec{v}_M = \vec{u} \times \vec{M}$ $\vec{v}$ vector, M a point on the edge, $\vec{u}$ the direction vector of the line, and $\vec{v}$ its other vector. Note that if a point M of the edge is on the line, its $\vec{v}_M$ vector is null. We then compute the sign of the dot product $\vec{v}_A \cdot \vec{v}_B$. If the sign is negative or null, it means that there is a point on the line and on the edge (in-between A and B), otherwise, it means that the intersection point between the line and the edges supporting line is not on the edge.

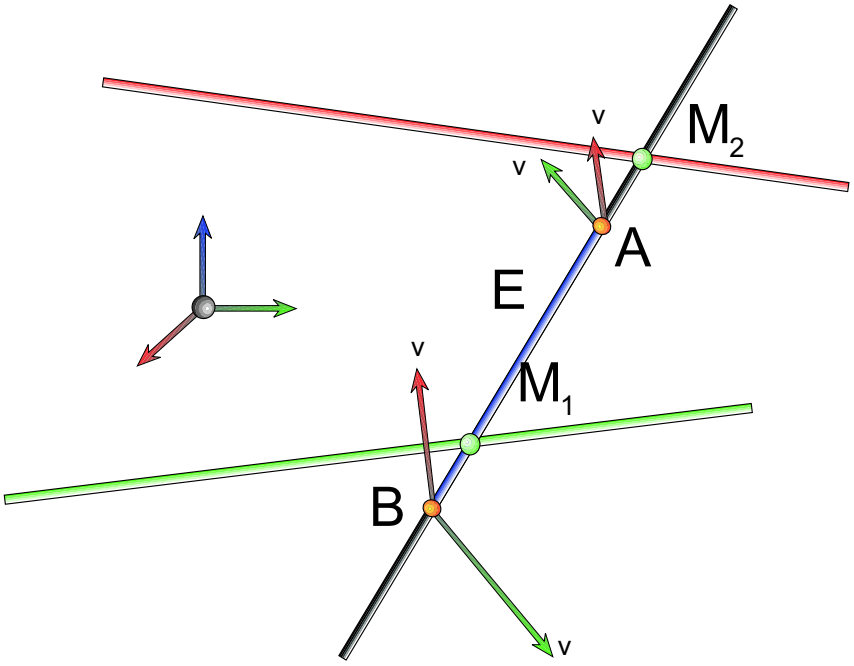


Figure 3.4: The on-edge test

Note on the degenerate configurations In the previous computations, we supposed we had a non degenerate configuration to be handled, but we need to test its degeneracy beforehand. A degeneracy is defined by a kernel dimension greater than 2. We first note that if two lines are at a distance below ϵ, then their side operator is below ϵ as well. When reducing the matrix using a Gauss pivot approach, we allowed column switching for getting a better pivot. If a pivot is below ϵ, it means that the side operator with the line in the image of the linear map is below ϵ. We thus exclude pivots smaller than epsilon and consider them as zeros for the reduction. The configuration with kernels dimensions greater than two is thus encouraged.

Occlusion Concerns

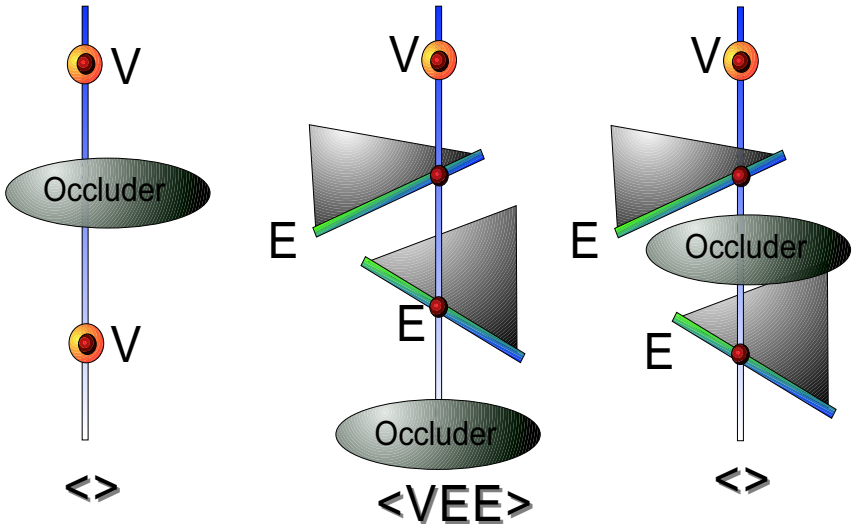


Figure 3.5: Illustration of the occlusion concern.

The main issues of visibility are to compute occlusions. The previous definitions of extremal stabbing lines and algorithms to compute them did not take occlusion into account.

A complete algorithm on occlusion for extremal stabbing lines validation is detailed in section 5.1. The main idea of the algorithm is to go forth onto the ESL, and to test if all the elements of the input set of generators for the ESL computation algorithm are hit before hitting the first blocking element. If not, then the ESL is said to be *not validated*, and is not taken into account.

3.3 Swath

Associated to the concept of ESL is the concept of **swath**. A swath is a continuous set of lines, which defines a critical line set regarding visibility. For example, for a given viewpoint, the set of rays from the viewpoint hitting an apparent boundary edge (which is called a **silhouette** edge) is a swath. It defines the visible boundary of an object and thus the limits of space it occludes.

Swaths are critical line sets of dimension one, and can be defined the same way as ESLs, either with the catalogue approach, since the catalogue implicitly defined swaths as connections between ESLs, either with an algebraic approach as a critical line set of dimension 1.

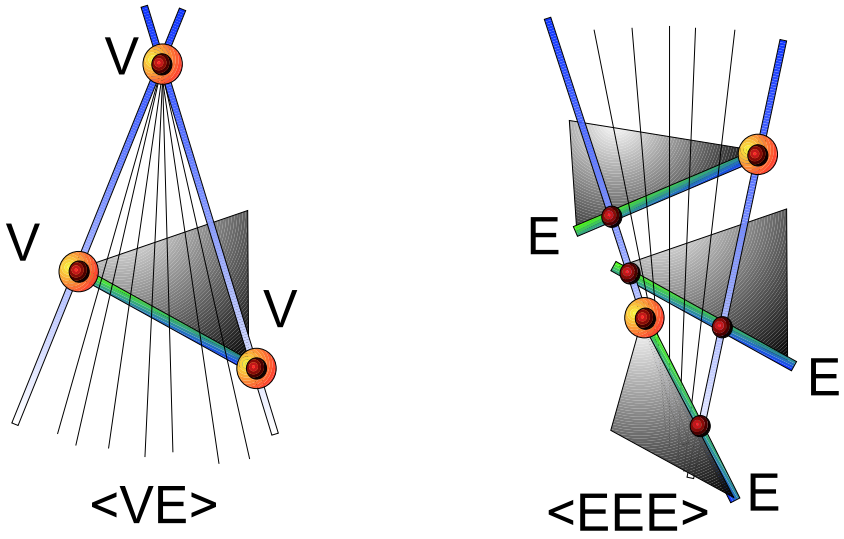


Figure 3.6: Illustration of a planar swath, and a non planar swath

Swaths and Shadows

As seen in the previous work, shadows casted by point light sources can be computed using the shadow volumes algorithm. Such a technique is a good example to illustrate swaths. The boundaries of shadow volumes are infinite polygons with ESLs originating at the point source, and a silhouette edge. A swath with generators the point source and the silhouette edge supports this infinite polygon. The shadow boundaries casted on receiver surfaces are the intersection between the infinite polygon and the surface, which is also the intersection between the swath and the receiver.

Swaths and Algebra

In the previous section, we presented ESLs in an algebraic approach. ESLs were defined as intersections between linear maps kernels and real and normalisation variety. Swaths may be defined the same way, but with a dimension restriction of one (instead of zero for ESLs).

In that definition, we can see that if two ESLs have linear maps in common so that the intersection of their kernels and real and normalisation variety is of dimension one, then the set of linear maps in common defines the swath (which is a set of generators). A direct consequence of this remark is that the two ESLs sharing **enough** generators (or linear maps by extend), are the boundaries of a swath.

3.4 Graph

Let us consider the following example : we want to study the set of ESLs and Swath originating at a viewpoint. This visibility query on the scene has many applications, such as the computations of shadows casted by a point light source.

In this example, ESLs are of kind VV and VEE, with first V being the viewpoint. Swaths are of kind VE.

All swath boundaries are ESLs of kind VEE for apparent vertices or VV for intrinsic vertices. Note that, in our approach, a swath is partitioned into sub-swathes if ESLs lie between its boundaries. Computing all these discontinuities in visibility results in a set of ESLs and swaths, in which swaths have ESLs as boundaries, are planar and ESLs are surrounded by swaths.

This structure can naturally be seen as a graph with ESLs being the nodes and swathes being the arcs.

This graph approach can be extended to other visibility queries, and is not restrictive to planar swaths. It has first been presented by Durand et al in [DDP97].

Chapter 4

Epsilon Visibility - Epsilon Predicates

In the epsilon context, every predicate described previously has to be exmained or redefined. Indeed, what is true with infinite precision arithmetic is not necesseraly true with a floating point arithmetic, and especially for 3D geometrical computations. For example, the alignment of three points in 3D space is not always well defined. Whether it be because of the points coordinates which are given in floatting point arithmetic and thus are not exact anymore, or the computations are done using a regular FPU which has finite precision.

4.1 Epsilon Criteria

Epsilon Contact

In section 3, we presented the hit criterion. An objet is hit by a ray if they share a point. In our epsilon approach, we want to keep this hit criterion even if the ray and the object do not exactly share a point. Also, some definitions required a vertex to be hit by a ray which is very difficult to insure using floating point arithmetic.

Our epsilon hit criterion is the following: A ray ε**-hits** an object if the distance between the object and the ray is bellow the predefined ε.

This definition is consistent with the previous for a null value for ε. Also, we use the term ε**-contact** for a configuration satisfying the ε-hit criterion.

By the way, as showed through figure 4.1, an object may be ε-hit by a ray for some value of ε, but not for another smaller value.

For the rest of the discussion, we suppose that a **fixed** value of ε has been given as input.

Epsilon Block

The redefinition of the hit criterion implies the redefinition of the generate / block interaction. The formal definitions are not exactly the same, but the key idea is similar. See figure 4.2 for illustration.

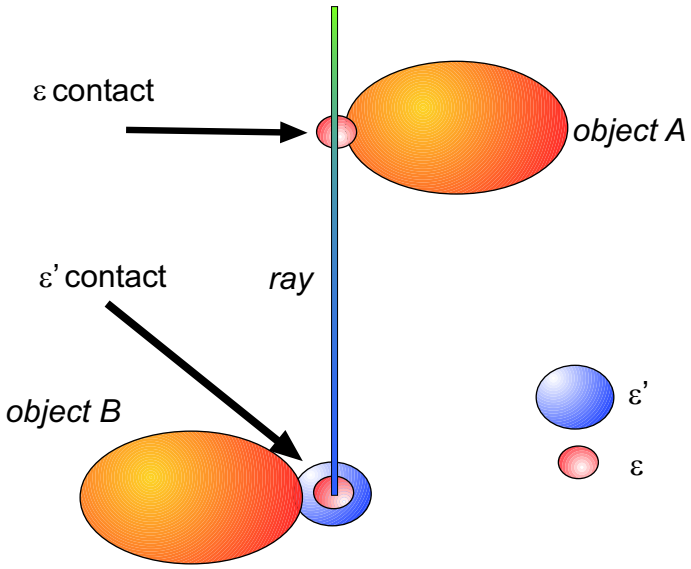


Figure 4.1: Illustration of the ε-contact concept

Let C_ε be a cylinder of radius ε around the ray. An object is said to ε-**block** a ray if there is no path around the object bound into the cylinder.

In the same way as before is defined the tangency criterion, getting a consistent binary ray classification.

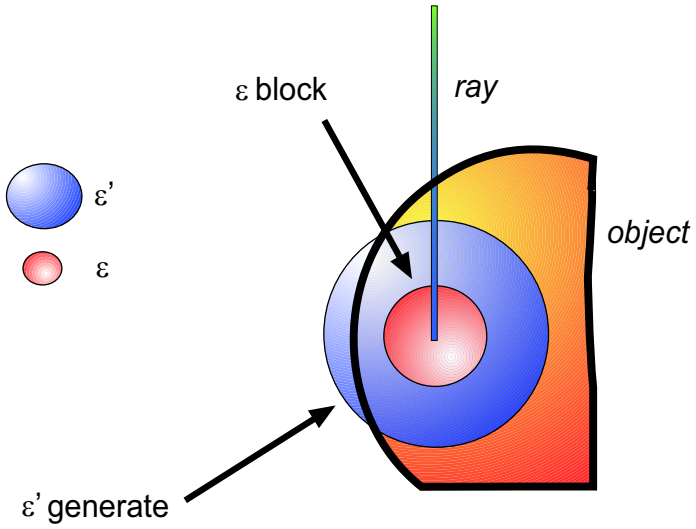


Figure 4.2: Illustration of the ε-block concept

4.2 Epsilon Predicates

As stated in chapter 3, we consider scenes made of polyhedra, with vertex, edge and face primitives. We present in this section the visibility predicates for such primitives.

Hit Criterion

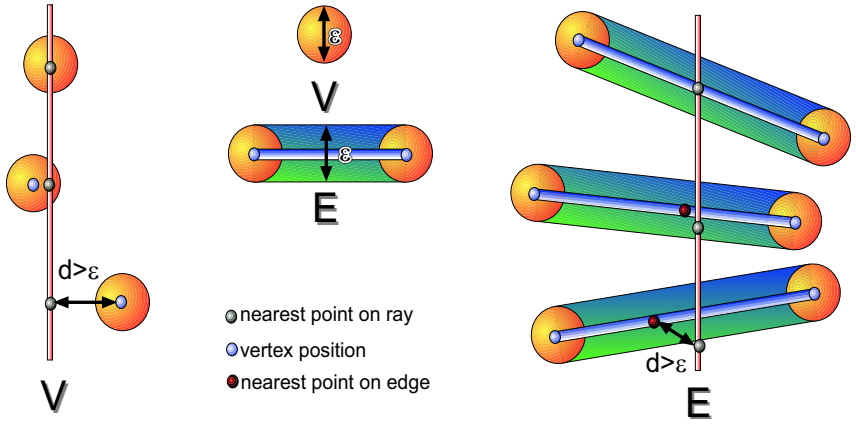


Figure 4.3: Illustration of the ε-hit criterion for vertex and edge

Vertex A vertex is ε-hit by a ray if the distance between the ray and the vertex is bellow ε. The distance function used is the usual Euclidean distance, which is computed in this case between the vertex itself and the nearest point on the line. The nearest point on the lined is obtained by orthogonal projection of the vertex on line.

See figure 4.3 for illustration.

This distance may also be obtained another way using Plücker coordinates, see Appendix A for details.

Edge The distance between an edge and a ray is computed between the two nearest points. These points are obtained by orthogonal projection of lines (ray and edge support) into the planes orthogonal to the line directions. This technique is only valid for Euclidean distance, which is the main motivation of our choice for this distance.

By the way, if the nearest point on the edge supporting line from the ray is not on the edge (segment between the vertices), then the distance is given by the distance to the nearer vertex.

Once the distance computed, the test is performed for ε-contact.

See figure 4.3 for illustration.

It is interesting to note that the edge is in fact a composition object, made of its segment and its boundaries, which are vertices. If a ray is in contact with either of this element, then it is in contact with the edge. This remark leads us to the ε-hit criterion for the face, which is also a composition object.

Face The interaction between a ray and a face can be of different kinds. These types of interactions are separated into the **regular** and **special** face hit configurations.

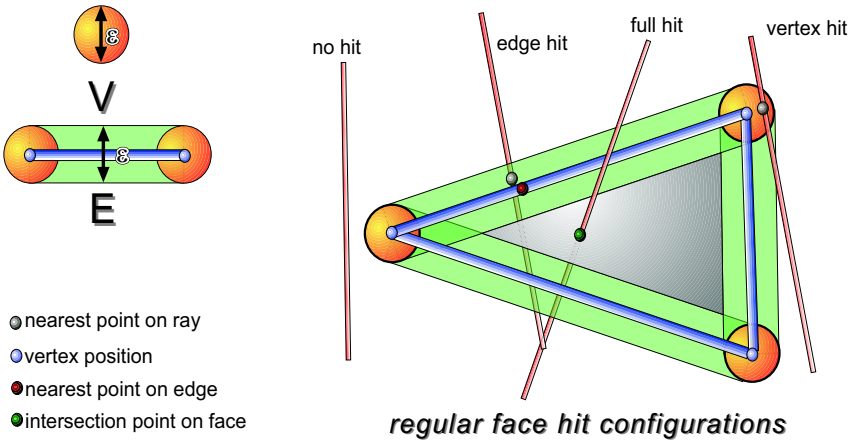


Figure 4.4: Illustration of the ε-hit criterion for a face - regular interaction

The regular face hit configurations are the one present without ε concern, that is (see figure 4.4 for illustration):

- no hit : the ray does not hit any part of the face
- full hit : the ray hits the face in its main frame, without hitting any boundary element
- edge hit : the ray hits an edge of the face (and no vertex)
- vertex hit : the ray hits a vertex of the edge

These configurations are well known and do not require further details. The hit criterion is straightforward and further predicates are described afterwards.

The special face hit configurations are the one introduced by our ε approach. They are the following (see figure 4.5 for illustration):

- non planar double edge : due to the fatness of the edges (cylinders) a ray may hit two edges which are connected by a vertex without neither being coplanar to the face (even almost), nor hitting the shared vertex.
- planar hit : this configuration could have appeared in the exact arithmetic approach. By the way, we still consider it special since there is no consistent way to distinguish between this configuration with two edges hit and the previous one.

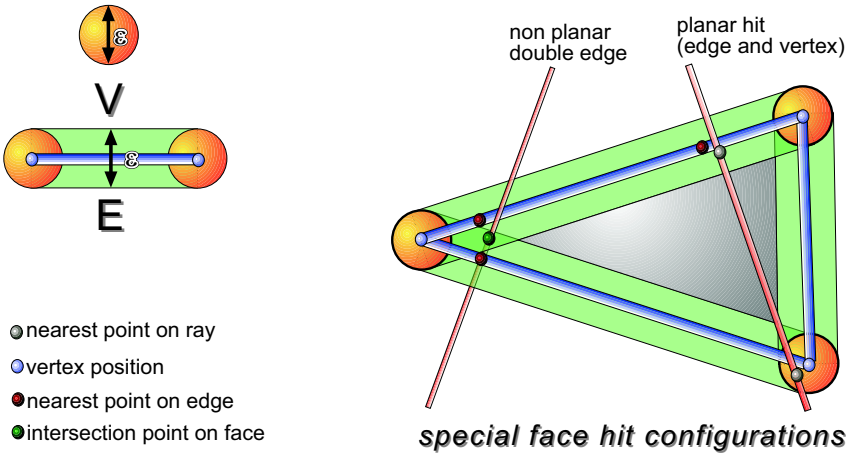


Figure 4.5: Illustration of the ε-hit criterion for a face - special interaction

Both these special configurations are handled the same way, using the multiface tool, described bellow in section 4.3. Besides, the hit criterion is still straightforward, block and generate predicates are detailed in the Multiface section 4.3.

In order to compute the interaction between a ray and a face, we apply the following algorithm:

face-hit algorithm

```
V = ∅
E = ∅
for each vertex of the face
  if hit, insert into V
for each edge of the face
  if connected to a vertex of V do nothing
  else if hit, insert into E
if #V = 0 AND #E = 0 then
  if hit main frame of the face
    return FULL-HIT
  else
    return NO-HIT
else if #V = 1 AND #E = 0 then
  return VERTEX-HIT
else if #V = 0 AND #E = 1 then
  return EDGE-HIT
else
  return SPECIAL-HIT
```

Figure 4.6: face-hit pseudocode

Vertex Block / Generate

Now the hit criterion has been established for scene elements, we have to define the Block / Generate test. Remember that these scene elements represent polyhedra and thus shall not be considered individually but as a whole; still local computations are sufficient. Besides, we make the assumption that we have connectivity information at our disposal. If not, please refer to section 5.1 for such configurations.

A vertex is a spatially localised point of a polyhedron, but is just another point for the underlying object. The neighbourhood of the vertex has to be considered for proper treatment of this Blocker / Generator test.

The idea of the Blocker / Generator test is to check if the ray grazes the object, that is if at this particular position, and from the viewpoint of the ray, the object is hit at a silhouette[1] point or not.

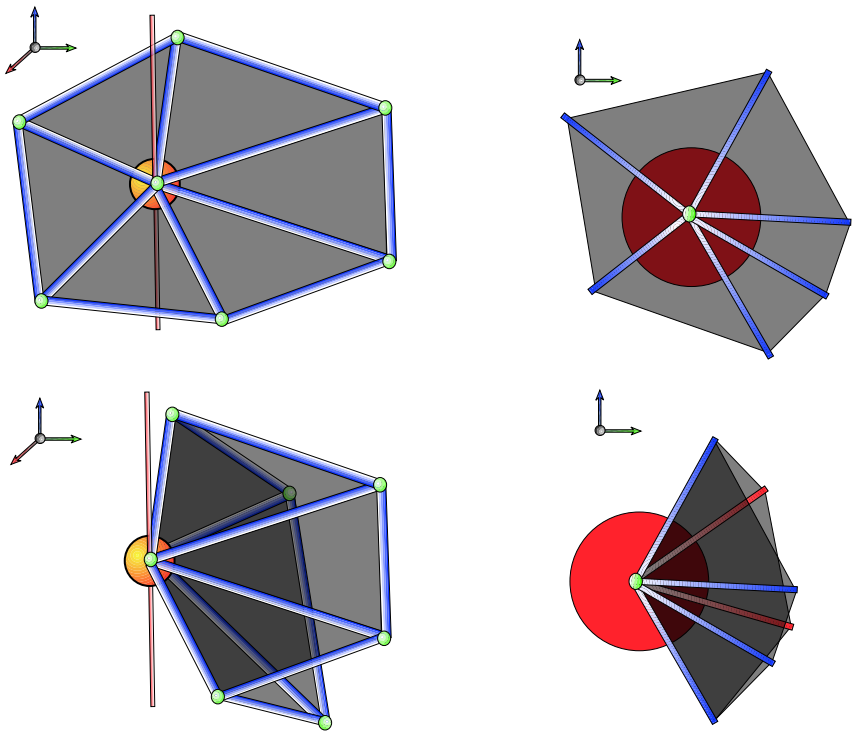


Figure 4.7: Illustration of the vertex block predicate. Left : 3D scene, right : projection on π plane. Up : the vertex is a **Blocker**, down : the vertex is a **Generator**

[1]The silhouette is the apparent boundary of an object from a given viewpoint.

To achieve this test, we orthogonally project the neighbourhood of the vertex (faces), on the π plane which is orthogonal to the ray. Then, in this plane, we compute the angular part around the vertex covered by surrounding faces. If this angular part covers the whole angular sector, then the vertex is a blocker, otherwise, it is a generator. See figure 4.7 for illustration. The red disc represents the whole angular sector. In the top example, the whole disc is covered, the vertex is a Blocker, in the bottom example, a part is uncovered, and the vertex is thus a Generator.

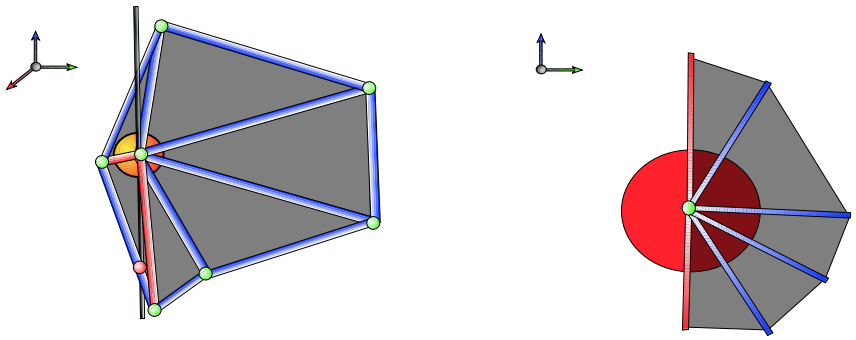


Figure 4.8: Illustration of the special hit for a face connected to a vertex for vertex Block / Generate test. The special-hit face is not taken into account for angular section covering

Besides, robustness issues may appear. For example, if a face is almost orthogonal to π, the result is unpredictable: on which side will the face be projected ? The angular portion covered by such a face is not the result of robust computations See figure 4.8 for details.

To avoid arbitrary results and ensure the robustness of our predicate, we apply the following filter on the faces: A face projects on π if no edge nor vertex of this face, other than the two edges connected to V, and V, is hit by the line, that is if the face is vertex-hit by the ray (and no special hit).

Note that this filter is consistent with the original definition of the Block / Generate test since if the face is special-hit by the ray, it means that there are free paths in the ray's ε-cylinder around the face.

Edge Block / Generate

The concept of this criterion is quite similar to the one for the vertex, but the algorithms differ. An illustration is given with the 3D scene and the projection on the same π plane, figure 4.9. The following algorithm only applies for any edge-hit and has only the direction of the line as input. The predicate is also a **silhouette** predicate.

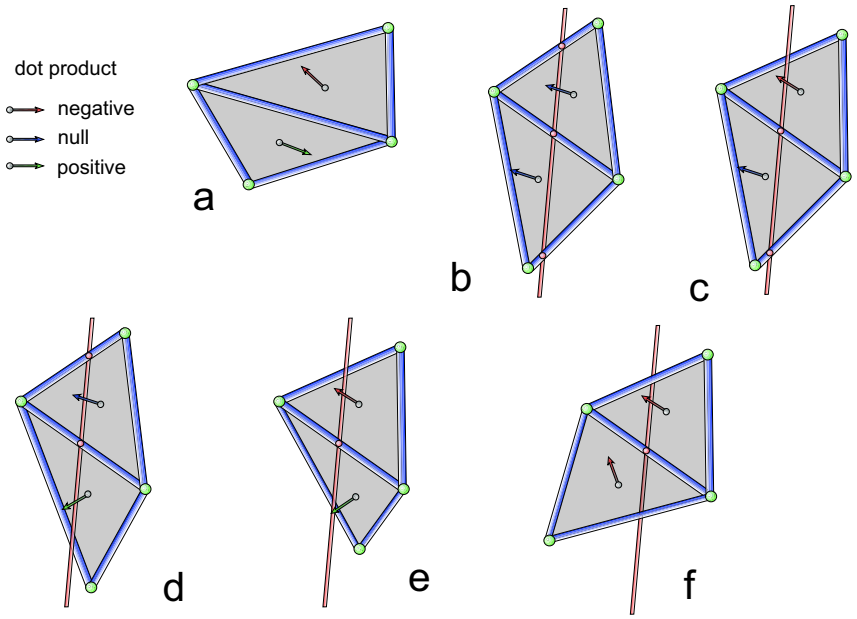


Figure 4.9: a, inconsistent normal orientation; b, flat edge; c, silhouette - flat; d, flat-silhouette; e, silhouette; f, block

The algorithm is as follows: Let $\vec{r}$ be the direction vector of the ray, let $\vec{f}$ be the normal to a face and $\vec{g}$ to the other. The normals must be consistently oriented, that is the normal continuously goes from $\vec{f}$ to $\vec{g}$ around the edge. Then, we compute the dot products: $s_f = \vec{f} \cdot \vec{r}$, $s_g = \vec{g} \cdot \vec{r}$. Then we have the following configurations:

- $s_f = s_g = 0$, the edge is flat and thus not silhouette.
- $s_f = 0$ or $s_g = 0$, the edge is flagged as silhouette, with a planar face.
- $s_f \cdot s_g < 0$, the edge is silhouette.
- $s_f \cdot s_g > 0$, the edge is not silhouette.

See figure 4.9 for illustration.

Note that this predicate does not take ε into account. In fact, the geometrical extend of the face connected to the edge would lead to inconsistencies for any ε-based silhouette predicate, since for the same angles, an edge connected to small faces would be silhouette whereas with a bigger face, it would not be silhouette. This kind of inconsistencies is not permitted in our approach.

4.3 The Multiface

As stated in the previous section, our ε model has introduced special hit configurations leading to the necessity of a specific treatment. Thus, for a special-hit face, a whole group of faces has to be studded in order to get consistent predicates. For example, sliver triangles or small (bellow ε spatial extend) triangles cannot be considered alone, and their neighbourhood has to be taken into account. This is the main motivation of the multiface tool.

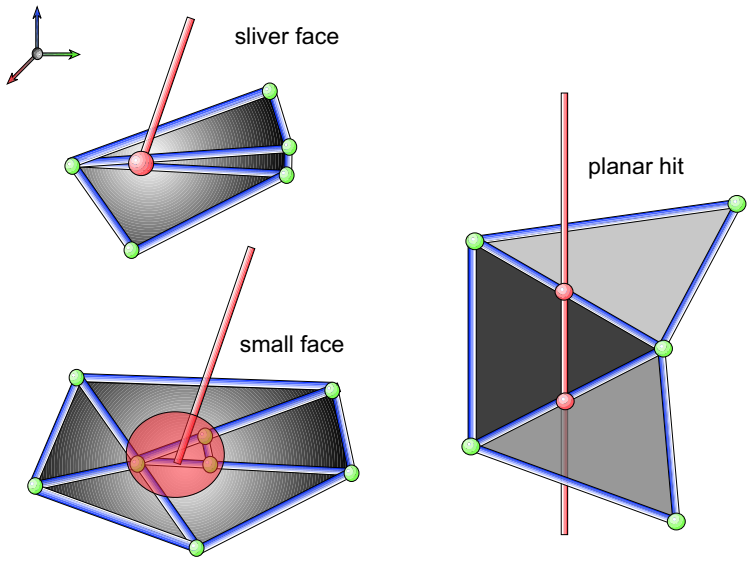


Figure 4.10: Example of configurations needing the multiface tool

The multiface is a technique related to our ε approach. The tool is only necessary for degeneracies or special configurations which arise from our ε predicates and criteria. Examples of such configurations are given in figure 4.10.

As hinted by its name, the multiface idea is to consider a set of face as a whole group, and run consistent predicates on the group instead of individual faces. The first part of this section is the construction algorithm followed by the Generate / Block predicate. Note that as far as the multiface is only used when a special face-hit is encountered, the hit criterion is not needed here.

It also should be noted that the multiface is a *volatile* object, which is valid for only a given ray, and the results of the predicates are given without returning the multiface structure. We thus suppose that the ray is fixed, and we have as input a special-hit face.

Construction

The construction algorithm needs connectivity. If connectivity is not available, then the results should not be what expected: the multiface will not work properly and set of faces (unconnected) will not block a ray they should block. For unconnected face treatment, please refer to the Blockerfan. Besides connectivity, the algorithm also need the silhouette predicate for edges.

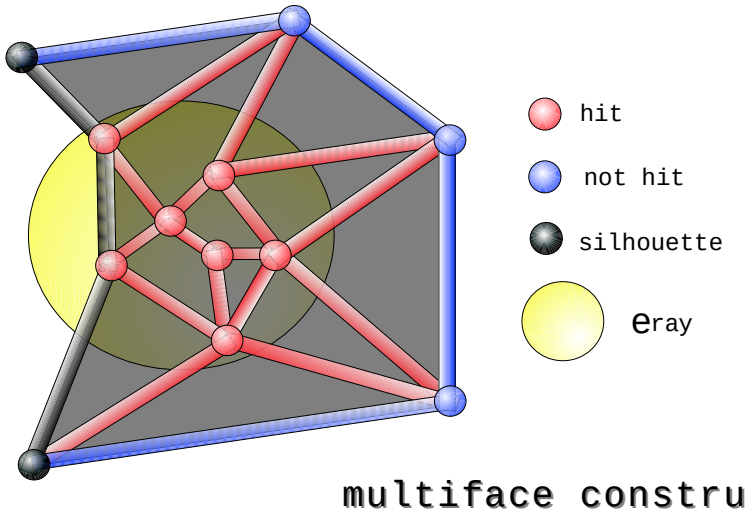


Figure 4.11: Construction of the multiface

The construction of the multiface is made from the special-hit face and around the ray, getting away from the first face. We start with the input special-hit face, and we add a connected face if the following conditions are fulfilled:

- the face is hit by the ray
- the connection between the face and the multiface is done by a non silhouette edge

We proceed all the connected faces this way, recursively. The algorithm stops when no connected (by non-silhouette edges) faces are touched by the ray. See figure 4.11 for illustration.

Figure 4.12, is given a complete pseudo code of the construction algorithm, in this pseudocode, the f^* set is the set of faces connected to f with a non-silhouette edge.

multiface construction

```
MF = {f}
C = f*
D = {f}
while C ≠ ∅
  pop g from C
  D = D ∪ {g}
  if g hit by ray
    MF = MF ∪ {g}
    C = C ∪ (g*  D)
end while
return MF
```

Figure 4.12: multiface construction pseudocode

Predicate

The Generate / Block predicate is quite similar to the one of the vertex. The same slice approach is used, but in a slightly different way.

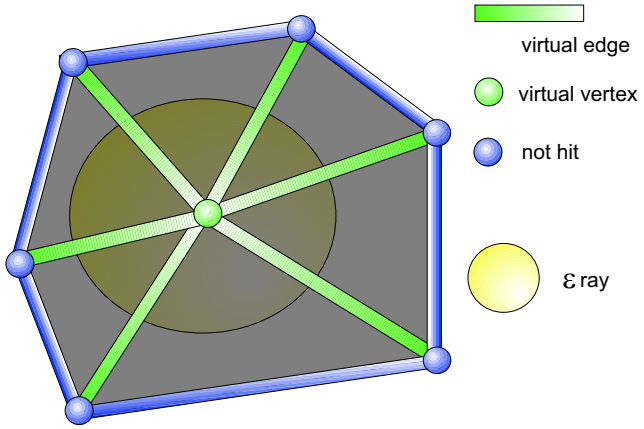


Figure 4.13: Block configuration

We project all elements of the multiface in the plane π orthogonal to the ray. The intersection point between π and the ray is Ω. We then get the boundary of the multiface which are edges either silhouette, or not hit by the ray. For each non hit boundary edge (can be silhouette or not), we build a *virtual* face (which will be a triangle) which is defined by the virtual vertex Ω, and the boundary edge. Additional virtual edges are drawn between Ω and the boundary edges bounds. We then consider this local virtual mesh as a vertex surrounded by faces, and we then apply the previous predicate. Note that elements hit by the ray do not contribute to the predicate as detailed as a special configuration in the vertex case. Two examples are given: figure 4.13 for block and figure 4.14 for generate.

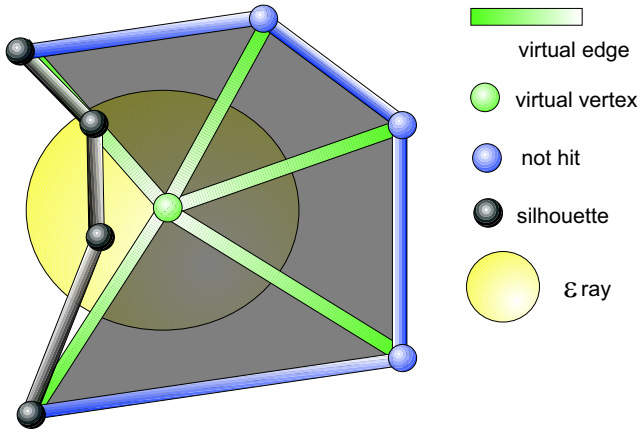


Figure 4.14: Generate configuration

4.4 Epsilon Visibility Complex

The visibility complex has first been introduced by Durand et al. [DDP02] for generic configurations. It can be extended to an epsilon visibility complex in the same way the visibility skeleton has been extended. However, on a topological point of view, the dimensionality of the events is not respected since the epsilon hit "fuzzy" criteria make the set of lines of a nD event go 4D. For example the set of lines through a point is 2D, but the set of lines through a ball of radius ϵ is a 4D algebraic variety.

Let us consider the set of oriented lines in space. If we consider the representation of lines presented in appendix A, this set of lines is an algebraic variety of dimension 4 embedded in a 6D space. This variety is the set of zeros of the ideal defined by the normalization quadric and the reality quadric. Studying subsets of oriented lines in space is equivalent to studying subsets of the variety.

The visibility complex is a graph in the algebraic variety of lines. For example in the very simple scene containing two disjoint spheres (see Figure 4.15, the set of free lines (stabbing no sphere) is a cell, the set of lines stabbing one sphere only is another cell (A and B), and the set of lines stabbing two spheres is another cell (A&B). Each cell is a variety of dimension 4. The boundaries of these varieties are sets of lines tangent to a sphere or two. The set of lines tangent to the two spheres is of dimension 2 (the two small magenta spheres on illustration). If we consider ϵ-events, the graph is no longer a graph but a set of cells. Each cell represents an event of dimension from 0 to 4. On Figure 4.15, is illustrated the set of cells for two different values of epsilon. On top, all the events of the complex (without the concept of ϵ-events) are present in the ϵ-complex. In the bottom, with a greater value of ϵ (greater than the spheres radius), some events have vanished. Higher dimensional events have been swallowed by smaller dimensional events. For examples every line stabbing a sphere is considered tangent to it.

We can thus make a remark on the size of the ϵ-complex compared to the regular complex: it is smaller !

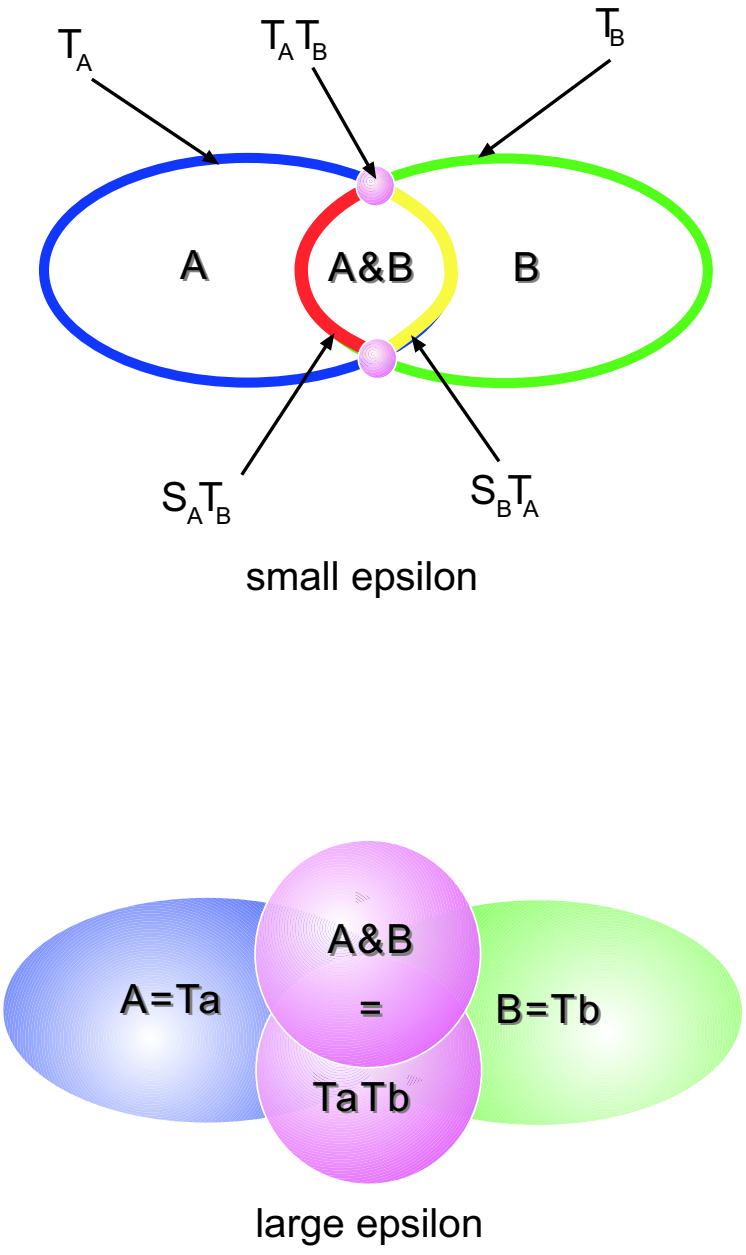


Figure 4.15: Illustration of the epsilon visibility complex

Chapter 5

Algorithms

Topics presented here :

- ESL casting
- Swath validation

5.1 ESL Casting

Basic ESL Casting

In the framework (chapter 3), we presented a way to compute generic ESLs, without taking occlusion into account. Since only a part of these lines will indeed be ESLs, the results of these algorithms are called **ESL candidates**. These candidates are **validated** or not through the ESL casting process which tests the occlusion of the candidates ESLs.

The ESL casting process takes as input the ESL candidate and the list of its **native** generators, that is the scene elements (vertices and edges), which were used to actually compute the ESL; also is given as input a starting point - or source point - for the ESL (the light source for example). This algorithm is similar to **ray casting**: the scene is traversed along the ESL from its source point to the first blocker encountered.

The ESL casting algorithm is provided in pseudocode 5.1.

ESL casting

```
N = native generators
S = ∅
begin at source point
while no blocker found
  go to next item hit
  if is blocker then
    blocker found
  else
    add item to S
end while
if N ⊂ S
  return valid
else
  return not valid
```

Figure 5.1: ESL casting pseudocode

BlockerFan

Introduction Some special configurations, especially objects in contact or special-hit faces, do not lead to a well-defined, or even any blocker. To prevent this inconsistency in the ESL casting process, another tool has been provided: the BlockerFan. This tool is used to gather blocking information along the ESL casting process to provide a well-defined blocker.

For example, let us consider the configuration drawn in figure 5.2. The first face encountered along the ray lead to a regular edge-hit, the edge being silhouette does not block the ray. The second face encountered is a special-hit since two of its edges are hit. This leads to the multiface but which does not extend since the edges are boundary edges, so the face does not block the ray. Finally, the last face is hit with a regular edge-hit, and thus does not block the ray either. A cut is provided on the right of the figure.

By the way, the underlying object should block the ray, and so should do the polyhedron. The BlockerFan tool is then used.

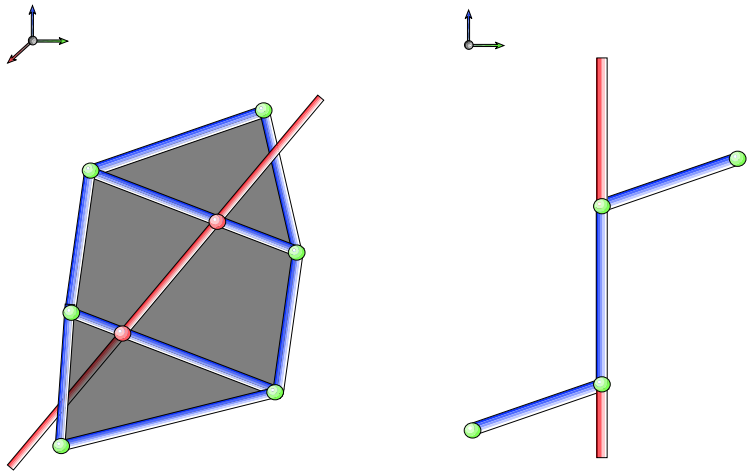


Figure 5.2: Configuration using the blocker fan.

Gathering The BlockerFan is a tool which gathers blocking information along the ray through the ESL casting process. As its name stands, it works as a fan of partial blocker. Each element encountered (whether is be a vertex, an edge or a face) has a contribution to the fan with a slice, or with depth. The angular sector around the ray, in the π plane is partially covered by partial blockers, and if enough slices are gathered to fill the whole pie, then the ray is blocked by the last element providing a slice.

For a better representation, imagine the ray as a cylinder on which you cut fat slices. If the cylinder is cut into two separate parts, then you found a blocker.

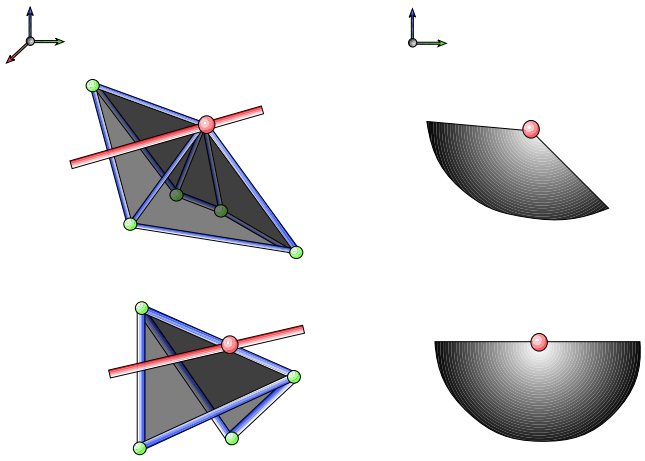


Figure 5.3: Computation of slices for vertex and edge

Slices are generated by edge of vertex hits. See figure 5.3 for illustration.

For each encountered element, a slice is added to the BlockerFan with a fatness set to 2ε, that is a spatial extend along the ray from the hit position minus ε, to the hit position plus ε.

Faces which are regular hit do not contribute the the blocker fan since either they are blockers themselves (which is the most frequent case hopefully), or they make a contribution through a vertex or an edge. Special hit faces behave differently: as illustrated in figure 5.4, the spatial extend along the ray is computed. An interval of positions along the ray are (which is fattened of ε, at each bound). Besides, slices may also be computed: if an edge of the face is not hit by the ray, it generates a slice of fatness, the spatial extend of the face on the ray. This slice is computed the same way slices are computed for the multiface.

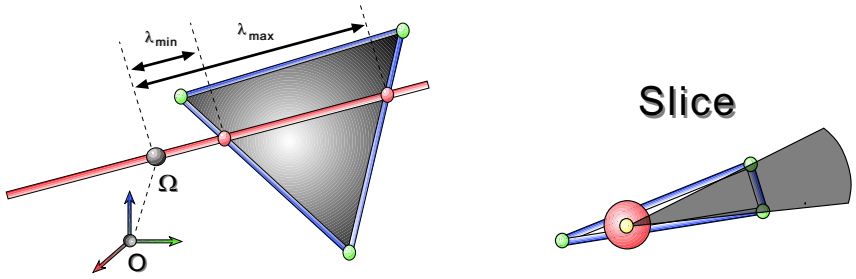


Figure 5.4: Computation of interval for face

This special contribution (the interval) will extend the previously encountered slices fatness, if they hit the face. That is for example, in figure 5.2, the first face, which is edge-hit will have a thin slice of size π (half the full pie), and the second face, planar-hit will contribute with a very small slice (planar hit), and a thick interval. This interval is in contact with the first face's slice, which is thus extended. Finally, the first slice will be in contact with the last one, and they will merge in a whole pie, blocking the ray at this last position.

To summarise, each scene element contributes to the BlockerFan with a thick slice. Faces will have thick slices, but with a reduced angular sector extend (slice of the pie in 2D - which can be null). Vertices and edges will have a thin slice, but with a certain angular sector extend.

A naive version of the algorithm would be to compute all slices and intervals and to extend the intervals in intersection. A progressive version of the algorithm is provided bellow.

The gathering algorithm is as follows: a pool of slices is build and updated at each object encounter. If at any update, the pool merges into a full slice, then the last object inserted is the blocker.

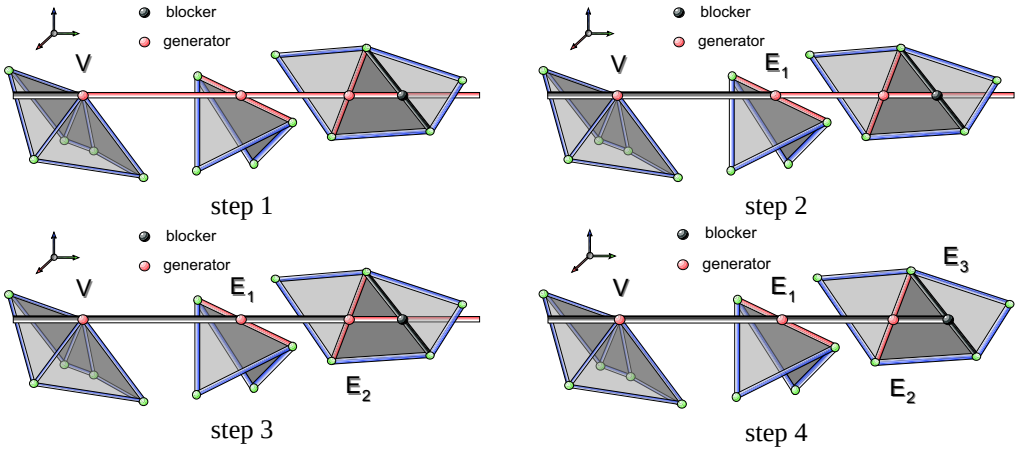


Figure 5.5: BlockerFan steps

If we reconsider the example given figure 5.2, detailed step by step in figure 5.5.

- Step 1, The first element encountered is a vertex, with a thin (2ε) slice. The pool is update with this element.
- Then, step 2, the second object encountered is at a further position, so the pool is updated by removing the slice of the vertex (too far), and inserting a new slice provided by the edge.
- Step 3, the third object encountered is an edge, the pool is updated by removing the inserted slice and inserting another one for the same reasons as before.
- Step 4, a face is special-hit, the interval is in intersection with the previous slice, and the slice of the pool is thus extended.

- Then finally a last edge is encountered at a position into the slices ray extend, and as the slices merge into a whole pie, this last element is a blocker.

As hinted by the algorithm, a progressive traversal along the ray of the scene is necessary for such an algorithm to work well. Also, the intersection positions of the elements on the ray in intrinsic coordinates is needed. These late algorithms are either detailed in appendix, or usual enough not to be detailed here.

It is important to note that even if the multiface is a good predicate for blocking for a face, it does not necessarily provide in a unique manner a blocker. It is often necessary to use the blocker fan to get such information. Even if the BlockerFan does not provide exact information for the blocker (especially for small or sliver faces), the result provided is at most at a distance ε of the exact result, which we considered satisfactory in our context - definition of ε.

Note that the BlockerFan provides a unique blocker whether it be a vertex or an edge (the face case being trivial since it does not need the blocker fan). And this will lead us to both consistent and easy to use results for the potential triangulation of the receiver.

5.2 Swath Validation

Swaths are built the same way as are ESLs. First, a **swath candidate** is proposed, and then validated, depending on occlusion computations. As introduced in section 3.3, a swath is a continuous set of lines between ESLs. This set may be defined by generators, in the same manner ESLs are.

Generic swaths are the following : VE and EEE in the same notations as ESLs. They are also subdivided into two types : **planar** and **quadric**. Planar states for a set of lines contained in a plane. Quadric states for the other configuration : the set of lines is a ruled surface. VE swathes are necessarily planar whereas EEE swathes my be planar or quadric.

A swath is proposed as a set of generators (VE or EEE), then the validation process is started resulting in one or several **sub-swaths**. Indeed, several ESLs may lie on the same combinatorial (in terms of generators) swath, that is for instance, let V be a vertex and E_1, E_2, E_3 be three edges, and E_1 is bounded by A and B and finally E_1 and E_2 appear to intersect from V as for E_1 and E_3. In this configuration, shown figure 5.6, the following ESLs might be build : VE_1E_2, VE_1E_3, VA, and VB. These ESLs lie on the same swath: VE_1. This swath will be cut into three sub-swathes, one between VA and VE_1E_2, the other between VE_1E_2 and VE_1E_3 and the last one between VE_1E_3 and VB.

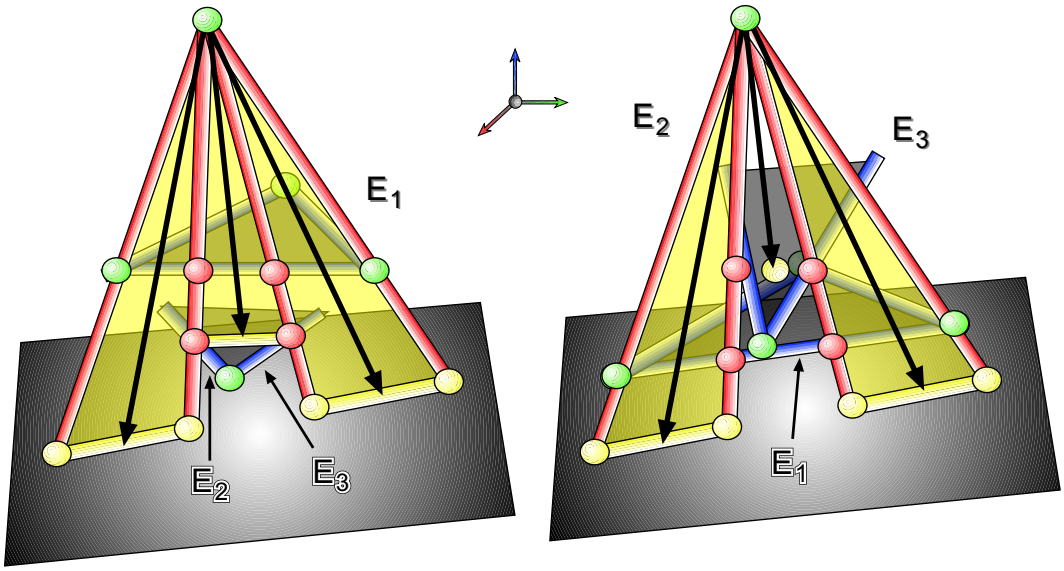


Figure 5.6: Swath validation illustration : left, three sub-swathes, all validated; right, three sub-swathes, the midline in the centre is blocked before reaching its generators, and thus eliminated

Once this partitioning is made, the occlusion tests are taken, and swathes are eventually validated or rejected. These steps are detailed in the following sections. First, generic swaths are studied, and

algorithm provided, then some degenerated swath, and finally the extension of swaths to 2D planar critical sets.

Swath partitioning

The swath partitioning algorithm is the following : for each swath proposed, all ESLs in the swath, that is a combinatorial approach : all ESLs hitting the generators of the swath, are listed. This set is then ordered as position of intersection along an edge of the swath (the reference edge). As planar swath may have an apex on an edge, and not necessarily on a vertex, the extend along all edges is computed, and the edge which is hit on the greatest extend by the swath is taken as reference edge.

Then, each intersection point between the ESL and the reference edge is computed, and ESLs are ordered by position of this point on the edge. Finally, ESLs are taken pairwise, with the nearest further one, and the swath is partitioned, exactly the same way the segment of intersection between the edge and the swath would be (see figure 5.7 for illustration).

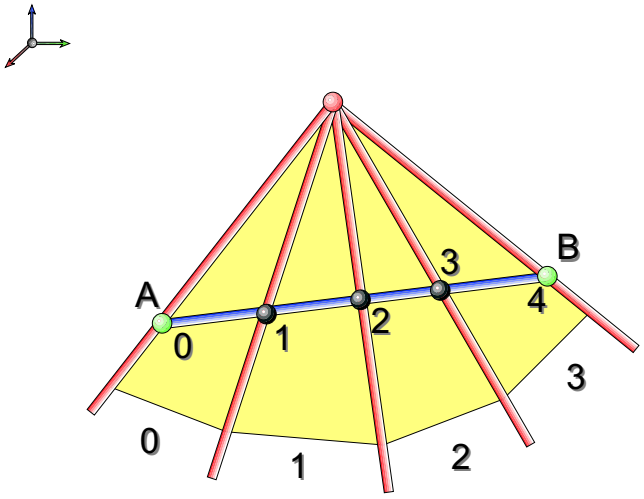


Figure 5.7: Swath partitioning illustration

Sub-swath Validation

It is important to note that as far as all ESLs are supposed to be computed and validated, all discontinuities in visibility along the swath are known and stored as the ESLs. This hypothesis is fundamental to insure the consistency of our algorithm.

The result of the previous remark is that visibility is continuous along each sub-swath, so that it can be sampled at any position of the swath, and the sampled *value* is constant all over the sub-swath. That is if all generators of the swath are hit by a line of the sub-swath which is not blocked

in-between, and has a blocker B, then all lines of the sub-swath hit all generators of the swath, and have the same down blocker B.

The validation process is thus straightforward : we sample visibility in the middle of the sub-swath (along a line we call the **midline**), applying the ESL-casting algorithm to the computed ray, with the set of native generators provided by the generators of the swath. Additionally hit generators are also stored and if the boundary ESLs of the sub-swath also hit these generators, then the set of generators is enriched by this last element. Such a swath is **degenerated**.

Midline Computation The midline has to be robustly computed in any configuration. We assume that each sub-swath, whether generic, or degenerated (as will be seen further on), has at least two distinct generators. This assumption is currently satisfied, and will not be overridden in the following section for degenerate configurations.

The computation technique for the midline is the following: we compute the best two points we know of the midline and get the line running through them. The best two points are given by the furthest midpoints on the generators. The midpoints are the barycentres of the two intersection points on the generators. Through these midpoints runs one and only one line. The alignment of these midpoints if proved bellow.

Midpoints Alignment Proof Let :

- $\delta_a = (\vec{u}_a, \vec{v}_a)$ and $\delta_b = (\vec{u}_b, \vec{v}_b)$ be the two ESLs bounding the sub-swath to be validated.
- $G = g_1, g_2, \ldots, g_n$ be the set of generators for the sub-swath, that is the intersection of the connexion generators of the two bounding ESLs.
- π_a be a plane orthogonal to $\vec{u}_a$, and π_b to $\vec{u}_b$.
- δ_b^* the projection of δ_b on π_a, and δ_a^* the projection of δ_a on π_b

The set of generators, projected on π_a (and π_b resp.) have an apex at the intersection of π_a and δ_a (resp. π_b and δ_b). The midpoints on the generators are aligned in each plane thanks to Thales theorem, and so in space if $\vec{u}_a \times \vec{u}_b \neq 0$, otherwise, the two lines being parallel, the midline is also parallel and runs through all generators, which are on the common plane.

The midpoints on each generator are thus aligned and can help computing the midline.

Extension to 2D Planar Critical Line Sets

Some configurations of edges may lead to special swaths: either overlapping or made of two planes. These configurations appear when two edges are coplanar, and have to be studied specifically.

We will study each possible configuration of coplanar edges configuration specifically, and propose a unified algorithm to treat them.

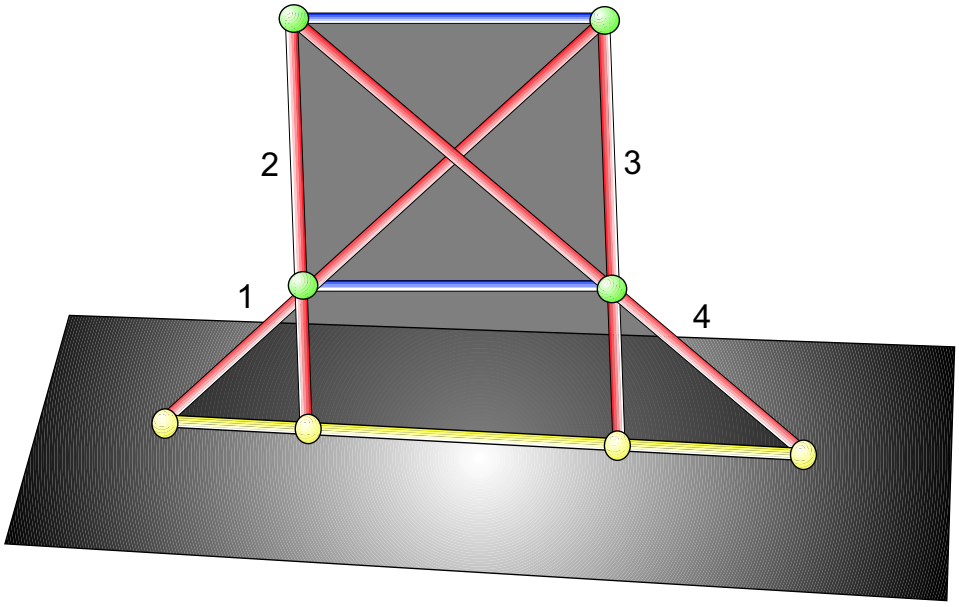


Figure 5.8: Overlapping swaths illustration - example of ordering

Overlapping Swaths As stated in [Hec92] and in [LTG92], if an occluder silhouette edge is coplanar to a source edge, then it generates a C^1 lighting discontinuity. In fact, in this configuration, several swaths overlap, see figure 5.8. Every critical line is indeed listed in this configuration, but the main problem is that on the receiver, for a given point on the shadow limit, two critical lines intersect the receiver at this point. This double information leads to robustness issues for triangulation of the receiver mesh. We thus want to describe these discontinuities in a proper manner to avoid such robustness issues.

Two-planes Swaths On the other hand, two edges of the same face are also coplanar. If the faces supporting plane intersects a source edge, this configuration will generate C^2 lighting discontinuities. In this case, as illustrated in figure 5.9, some swaths are made of two parts, on two different planes, but no extremal stabbing line between them. We thus create what we call a **pseudo-ESL** at

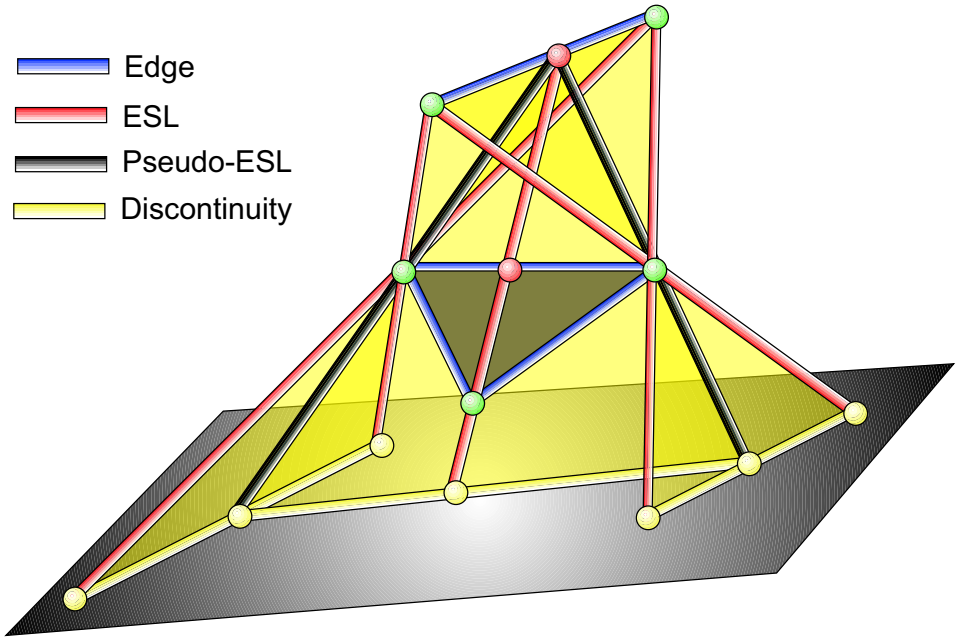


Figure 5.9: Illustration of pseudo-ESLs

the intersection of the two planes, in order to give bounds to the underlying planar sub-swaths of this two part swath. This pseudo-ESL will be treated exactly the same way as a regular ESL, in terms of validation, but enumerated specifically.

2D Planar Critical Line-Sets Partitioning In these configurations, the swath partitioning will not be made the same way as for the generic configuration.

In the Two-planes swath configuration, the partitioning is almost the same, since the only difference is that the sub-swaths are bounded by non-ESLs, but the approach is identical.

In the other case, the result depends on the receiver. In fact, to avoid robustness issues during triangulation, we give for constraints a polyline on the receiver which is the intersection of the swaths and the receiver. The vertices of the polyline are intersections of ESLs with the receiver, and the segments connect nearest vertices pairwise. Note that this is a 1D problem.

To achieve this polyline creation, we compute the intersections positions of the ESLs on the receiver, and we sort the ESLs in order of increasing position along the intersection line of the swaths plane and the receiver, see figure 5.8. Then, we get the ESLs pairwise, which gives the swath partition. The swath generators set is the intersection between the ESLs generators sets of the boundary ESLs, these sets being extended by edges connected to vertices of the original set. In the configuration of figure 5.8, swaths $(1, 2)$ and $(3, 4)$ are EV kind, and swath $(2, 3)$ is EE, which is

not enough to specify a 1D critical set. The latest swath is bounded by two coplanar ESLs which can define an apex for the swath, at their intersection point.

Chapter 6

Lighting

In the previous sections and chapters, we presented algorithms to compute and validate ESLs and swaths. These discontinuities have to be identified from the set of scene elements. We need to enumerate combinations of scene elements, which can give ESL candidates or swath candidates, and maybe ESLs and swathes.

This section is divided into several subsections: the first one states the relationship between shadow boundaries and visibility events, the second one describes shadow discontinuities generated by contact and intersections of objects (which are D^0 discontinuities). The third part describes an application to compute sharp shadows (point or directional light sources), and finally the last one for soft shadows (from area light sources).

For each application subsection, a naive algorithm is given, followed by optimisations. Besides, Algorithms given here only concern enumeration of candidate ESLs and swath, no other scene traversal or clustering optimisation technique is provided here. For such optimisation techniques, the reader should refer to ray-tracing related optimisation techniques, since the scene traversal and related algorithm are only needed for fat ray casting.

As in [DDP97], the ESLs, and swath are gathered and stored into a graph structure: ESLs are stored as nodes, and sub-swath as arcs.

6.1 Shadows and Visibility Events

This section describes the link between the radiance function describing lighting and shadows, and the visibility events described in the previous chapters. These paragraphs are inspired from [Hec92] and [LTG92].

The Radiance function

Global illumination problem is often formulated using **radiance** functions. These functions represent the energy flux leaving a surface, originating from the surface itself or re-emitted from other surfaces. An example of formulae for a radiance function (for surface i, at position x, with lambertian surfaces) is given by:

$$L_i(x) = L_i^e(x) + \rho_i \sum_{s_j \in S} \int_{x' \in s_j} L_j(x') \frac{\cos\theta_i \cos\theta_j}{r^2} v(x,x') dA(x')$$

Where

- L_i^e is the emitted flux
- ρ_i is the Bidirectional Reflectance Distribution Function constant in lambertian context
- $v(x,x')$ is the visibility function between x and x': 1 if visible, and 0 if not
- $dA(x')$ is the differential area element centred at x', on s_j
- r is the distance between x and x'
- θ_i and θ_j are the angles between the surface normals and the line connecting x and x'

In the expression of the radiance function, we can find the visibility function: $v(x,x')$. We note $V(x,s_j)$ the part of s_j visible from x. The integrand in the radiance function can thus be split into two parts, one being equal to zero (since for x' out of $V(x,s_j)$, the visibility function equal to zero. The radiance function can thus be rewritten:

$$L_i(x) = L_i^e(x) + \rho_i \sum_{s_j \in S} \int_{x' \in V(x,s_j)} L_j(x') \frac{\cos\theta_i \cos\theta_j}{r^2} dA(x')$$

If we assume that the radiance function is smooth over light sources, and that x is not on the light source, then the integrand is also smooth. Discontinuities in the radiance function originate from discontinuities in the visibility, that is along boundaries of the $V(x,s_j)$ function.

We will study discontinuities in the visibility function, which apply discontinuities in the radiance functions. A similar discussion first appeared in Heckbert's PhD thesis [Hec91], we therefore use the same terminology. If the radiance over the sources is smooth, the radiance function may have D^0, D^1 and D^2 discontinuities; a D^k discontinuity is where the function is $C^{k\ 1}$, but not C^k.

D^0 Discontinuities

These are discontinuities in the function itself. They either originate from sharp shadows, that is from point sources; or they originate from occluders lying on the receiver, that is contacts or intersections between occluders and receivers.

In the first case (point sources), critical swaths studied in the previous chapters define the limits in space of visibility or occlusion of the source. The D^0 discontinuities in the radiance function thus lie on these surfaces. Locating such discontinuities can be made by intersecting the swaths with the receiver (as is made by the graphics hardware with the shadow volumes technique).

This application is studied in section 6.3.

The second case is different, since such discontinuities appear whatever the source shape is. These discontinuities are due to the presence of geometrical special configurations (contact or intersection), and can thus be identified as a preprocess, independently from the lighting of the scene.

Intersection and contact elements are studied in section 6.2.

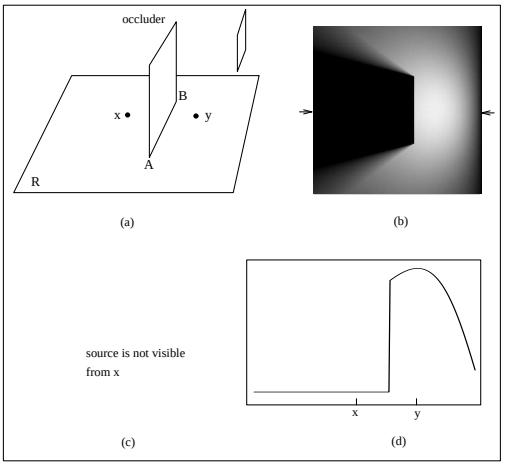


Illustration of D^0 discontinuities
- taken from [LTG92]

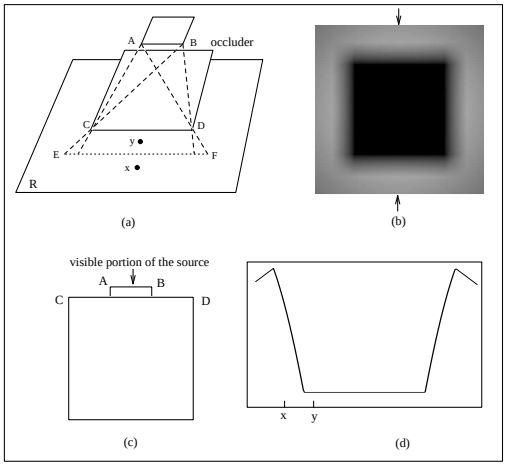


Illustration of D^1 discontinuities
- taken from [LTG92]

D^1 Discontinuities

These discontinuities come from degenerate configurations. In section 5.2, we saw through a pair of coplanar edges runs a two dimension set of lines, which are all contained in a plane. this plane is the locations of D^1 Discontinuities in the radiance function, since, on one side, the source is not visible, and going to the other side, the visible area of the source will grow linearly, leading to a D^1 discontinuity.

These discontinuities are identified by edges coplanar to source edges. They are studied in section 6.4.

D^2 Discontinuities

These discontinuities are the most common discontinuities. They run along the swaths described in the previous chapters, when generators are in generic configuration (that is except for the D^1 case just above). Note that for two coplanar edges, discontinuity in the radiance function is of order 1 only if one edge is a source edge !

D^2 discontinuities may be umbra or penumbra boundaries as well as inner penumbral discontinuities. They lie along swaths either planar or quadratic.

These discontinuities are studied in section 6.4.

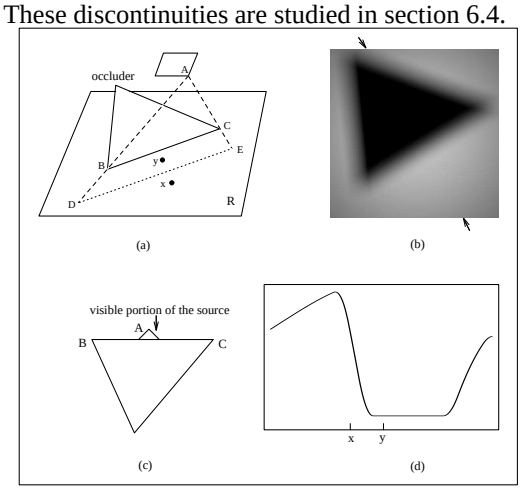


Illustration of D^2 discontinuities, planar - taken from [LTG92]

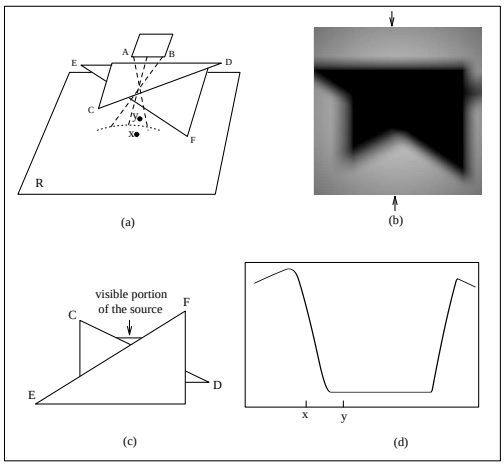


Illustration of D^2 discontinuities, quadratic - taken from [LTG92]

6.2 Intersections and Contacts

Introduction

As stated in section 6.1, D^0 discontinuities in the radiance function originate in contact of objects and intersections. Such features can be identified independently from the source shape and position.

In order to get consistent and robust identification of such elements, the same way we did for visibility events, we compute contacts and intersections, with the same ε threshold. The definitions of contact must be consistent with the ones given for ESL-casting. More precisely, an object at a distance bellow epsilon of another is said to hit it.

Besides, most intersection computations techniques need re-meshing. The aimed application is not to compute the intersected meshes for a further use, it is to compute visibility events on such meshes. We thus do not need to re-mesh the input geometry, but only to store locations of such discontinuities.

Note that intersections and contact elements are vertices (edge and face intersection, or vertex and face contact, in generic configurations), and edges (edge and face contact, or face and face intersection, in generic configuration); faces in contact do not imply any computations, since they do not imply any visibility discontinuity.

We compute a virtual mesh, we call the i-mesh. Such a mesh is made of i-vertices and i-edges, which are described bellow. These elements are computed and stored in a separate structure and do not imply any change in the input mesh. The visibility requests on the mesh are filtered to take this additional structure into account avoiding imprecise and unrobust computations due to re-meshing.

Each i-element as a structure, which holds references to elements in interaction (contact or intersection). For each i-element, depending on its origin, is given a structure, and an algorithm to identify its instances in the scene.

i-vertices

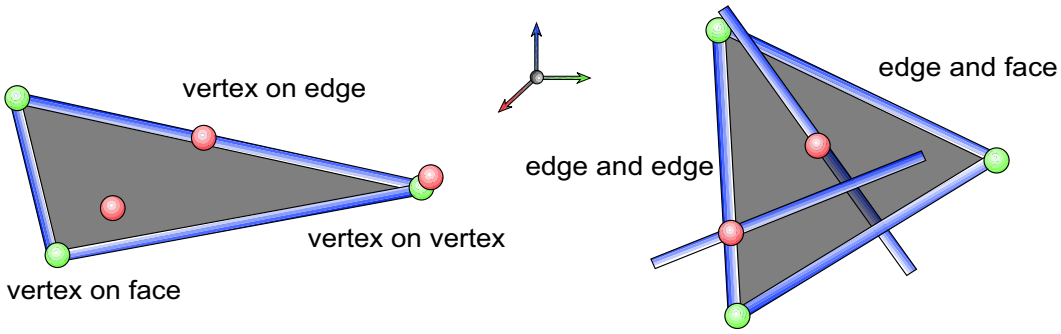


Figure 6.1: i-vertices from contact (left) and intersections (right)

contact i-vertices from contact, that is a vertices which lies on another face, edge or vertex (see figure 6.1), have the following structure:

- vertex
- face / edge / vertex

The precise location of such an element is given by the vertex coordinates for face and edge configuration, and at the middle point of the two vertices in the other case.

An algorithm to identify them is to compute the distance between vertices and other scene elements, using an acceleration structure (such as an octree), to only test elements which can be at distance bellow ε.

intersection i-vertices from intersection originate from edges in intersection with faces or edges, see figure 6.1. They have the following structure:

- edge
- face / edge
- point of intersection

The location of the vertex is explicitly given by the coordinates of the intersection point.

An algorithm to identify them is to cast a ray on the edges supporting line from a vertex bound of the edge, stopping at the other. All encountered elements whether it be edges or faces, generate such a structure. The intersection point is computed on the edge for the face-edge case (generic), and in the middle of the two edges nearest points in the edge-edge configuration (degenerated).

i-edges

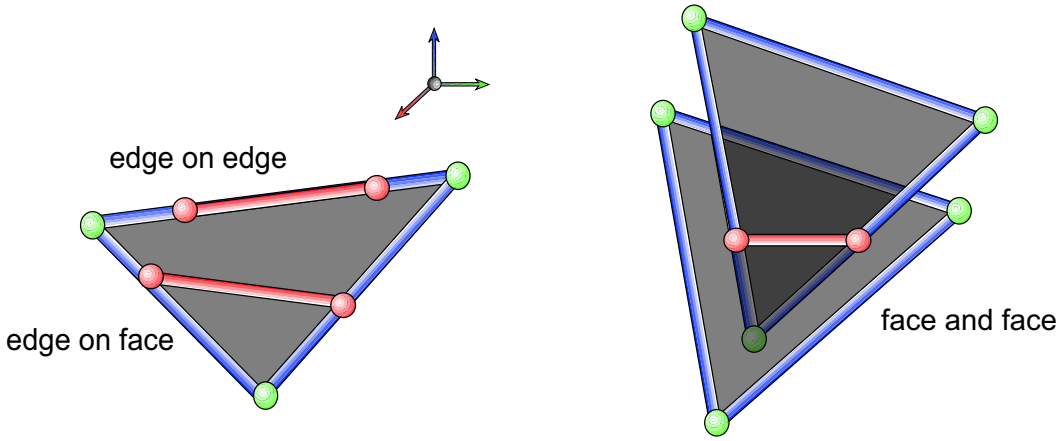


Figure 6.2: i-edges from contact (left) and intersections (right)

i-edges are segments lying at the intersection or contact of scene elements, but behave exactly as regular edges with respect to i-vertices, that is: an edge is a line segment between two vertices. i-edges are thus computed from the data given by i-vertices, which are considered computed at this point.

contact i-edges from contact originate at contact of edges on faces or edges, see figure 6.2. They are bounded by contact i-vertices which either lie on the face or on one of the faces edges. The structure is the following:

- edge
- face / edge
- i-vertices bounds

An algorithm to identify them is to combinatorially identify the i-vertices generators for vertices on a given face, and its boundaries. The main benefit of such an algorithm is that it is local.

intersection i-edges from intersection originate at the intersection of two faces, see figure 6.2. They are bounded by intersection i-vertices or contact i-vertices. The structure is the following:

- face
- face

- i-vertices bounds

The same algorithm as above is still valid to identify such elements.

6.3 Sharp Shadows

This application computes sharp shadow limits from point or directional light sources. All computed visibility discontinuities are light discontinuities and define sharp shadow limits. This application can be seen as an optimised (in terms of acuity of elements) version of the shadow volumes.

The **input** of the algorithm is a light source (point or direction), and a polygonal scene made of connected (or not) elements.

The **output** is the complete list of shadow discontinuities represented by swaths with the apex at the source, and blocker as shadow boundary receiver.

The source, whether it be a point light source or a directional light source, is considered as a native generator for the ESL candidates. Any other visibility discontinuity of the scene is not computed. Hence, ESLs originate at the source position, and swath are planar and their apex is at source position.

For this application, the graph built is topologically equivalent to the apparent boundaries of the scene objects viewed from the light source.

Naive The construction of the graph is divided into four steps:

1. ESLs enumeration
2. ESLs validation
3. swaths enumeration
4. swaths validation

The first step is purely combinatorial. All potential ESLs (generic) are computed and stored for further validation. The second step is straightforward: for each enumerated ESL, take the validation test, and if success, store the ESL as a node.

The third step is also purely combinatorial: potential (generic) swaths are computed and stored for further validation. The fourth step is also straightforward: for each enumerated swath, take the validation test, and if success, store each sub-swath as an arc, and connect the arc to the nodes corresponding to the ESLs, boundary of the sub-swath.

First step: generic ESLs enumerated here have to hit the source, which is considered as the first generator. This element behaves the same way as a vertex: for point light source, it is trivial; for direction light source, the light source constrains the direction of the line, which is two degrees of freedom our of the four available.

Generic ESLs are then SV and SEE for *source-vertex* and *source-edge-edge*. The enumeration algorithm is the simple enumeration of vertices and pairs of edges.

Third step: same remark, generic swaths are SE, necessarily planar, and enumeration is the enumeration of the edges of the scene.

The combinatorial complexity of the enumeration is $O(n^2)$, which has to be multiplied by the complexity of the ESL-casting process which is, in a naive approach or in bad configurations, linear; resulting in a complexity of $O(n^3)$.

Besides, we do not address the problem of ESL-casting here, but as far as it is only defendant on the intrinsic complexity or the scene and the configuration; it is an independent parameter for enumeration. These two aspects of the whole algorithm are completely separated. And we only work on the $O(n^2)$ complexity algorithm which is the pure enumeration, without any validation or casting concern.

Nested Calls The first remark is that the second and fourth steps do not need to be separated from the first and third one. As far as each ESL enumerated will be validated, the ESL casting process can be called into the enumeration process, storage being saved since number of candidates are not validated for occlusion matters.

This remark is valid for any other enumeration, and the implied optimisation is considered as always used.

Silhouette Elements This optimisation will not change the formal and theoretical result of our algorithm. Besides, in most cases, this optimisation reduces drastically time consumption.

The idea of this optimisation is very simple, from the light source, edge which are not silhouette edges do not cast shadow in the scene. Indeed, silhouette edges are the apparent boundaries of the objects of the scene, and thus are the only element casting shadows on other elements (may be on the same object if not convex). So, in the above enumerations, for swaths, only silhouette edges are taken into account. Thus, as a direct consequence, vertices which are not connected to a silhouette edge cannot be boundaries of swath and thus cannot generate ESLs. The number of vertices in enumerations is also reduced.

By the way, VEE ESLs are made of a silhouette edge, necessarily, and also another edge, which can be non silhouette. The enumeration of such ESLs is thus made of one silhouette edge, and another edge, which reduces drastically the amount of edge pairs (in nice configurations).

Swath Casting Once we chose a first edge, a silhouette edge, for the computation of the VEE ESL candidate, only edges which intersect the line set made by the source, and the silhouette edge, can contribute for an ESL candidate.

Another optimisation is thus possible using an acceleration structure. Suppose that edges of the scene have been stored in an octree structure by their position in space. An edge is referenced in an octree cell if it hits such a cell.

For a given silhouette edge, we compute three planes: one given by the source and the edge (noted π), and two others perpendicular to π and containing the source and a vertex of the silhouette edge (noted π_a and π_b). Then an edge may contribute for an ESL candidate only if it hits the plan π on a point I which is on the good side of both π_a and π_b.

To enumerate such edges in an optimised manner, we apply the following recursive function on the octree nodes (see figure 6.3).

This enumeration technique also reduces the number of tested elements drastically. Indeed, all elements listed with the algorithm lead to an ESL candidate, which are ESLs besides occlusion.

Final Algorithm The final algorithm is given figure 6.4.

octree enumeration

```
potential (N)
  E = ∅
  for each C child of the node
    B is the box of the child cell
    if B ∩ π = ∅ stop
    if B on bad side of π_a stop
    if B on bad side of π_b stop
    E = E∪ potential (C)
  end for each
  return E
```

Figure 6.3: octree traversal algorithm for VEE ESL candidate enumeration

Final Algorithm

```
Let
  S the source
  E the set of silhouette edges, in an octree
  V the set of vertices connected to S
ESL
  for each V ∈ V
    ESL-cast (SV)
  end for each
  for each E_1 ∈ E
    build π, π_a and π_b
    for each E_2 in octree traversal
      build SE_1E_2
      ESL-cast (SE_1E_2)
    end for each
  end for each
swath
  for each E ∈ E
    build H, set of ESLs in SE
    sort H along E
    η = first in H
    for η_+ ∈ H, ordered
      λ = midline (η , η_+)
      ESL-cast (λ)
        store (η , η_+) if success
      η = η_+
    end for, ordered
  end for each
```

Figure 6.4: Final graph construction algorithm

6.4 Soft Shadows

In this section, we describe an algorithm to compute shadows casted by area light sources, that is with non null spacial extend. This section is subdivided into 4 parts:

1. Visibility events and shadow boundaries
2. ESLs enumeration
3. pseudo-ESLs enumeration
4. Swath enumeration

Visibility Events and Shadow Boundaries

We presented in section 6.1 the link between visibility events and shadow discontinuities. We proved that each discontinuity in the radiance function lies on a visibility event, that is on a swath. Besides, some discontinuities of the radiance function, e.g. inner order two discontinuities, do not contribute to a visual change. Also, it is important to note that for display, the lighting is sampled, and then linearly interpolated between samples. The sampling has to be fine enough to represent visual changes of strong energy, but not too fine. In [CF90] is described

We make the following choices:

- only boundary shadow discontinuities are computed
- the shadow area is sampled depending on the irradiance gradient

Note that with these assumptions, we do not get the exact soft shadow boundaries, but we get an approximate result at a reasonable cost. Still every discontinuity in the radiance function can be computed using a naive algorithm enumerating every possible ESL and swath. of the scene, from the light source. But this approach is far too expensive (for average complexity scenes) to be detailed here.

The configuration here is similar to the case of sharp shadows. That is the elements of interest are boundaries in the radiance function discontinuities. These boundaries are given by lines which run through silhouette elements of the scene. But in this case, the silhouette criterion is not necessarily satisfied from each element of the source. There are two kinds of boundary elements: between the umbra region and the penumbra region which we call the umbra boundary; and between the penumbra region and the lit region, which we call the penumbra boundary.

Durand and al [DDP97], presented the Visibility Skeleton; a structure storing visibility information, and especially extremal stabbing lines and critical line sets (swaths). We present here an algorithm building parts of the skeleton on demand. It can be seen as a lazy approach of the Visibility Skeleton. To compute such a structure we need to enumerate and validate extremal stabbing lines and swaths. Once computed, these elements are used for meshing and lighting the scene.

ESLs enumeration

In [DDP97], a complete enumeration of the ESLs was performed; in our case, we only focus on ESLs which lie on a source boundary. Inner ESLs do not contribute to relevant irradiance discontinuities, and are thus not computed. We compute ESLs which have a generator on the source, that is:

- V_sV, V_sEE
- E_sVE, E_sEEE

The enumeration of V_sV and V_sEE ESLs is the same as for point light source, with silhouette optimization.

E_sVE are enumerated in the following way: we consider an edge of the source, say E_s. Then, we get a vertex of the scene, say V. We build a VE_s structure exactly the same way we did for V_sE, but the polygon is now made of two parts and the apex is not on the source. We still use this structure to identify edges which can help generate en ESL. These are the edges hitting this polygon. We thus have an ESL candidate, and perform the usual validation.

E_sEEE are more complicated to enumerate. In order to do that, we first select the source edge E_s and another edge which can be silhouette for E_s, noted E_1. We use a special structure called the hourglass, first introduced by Durand et al in [DDP97]. But we use this structure differently: the hourglass is computed, and then, we use an octree traversal algorithm bounded to this volume. This algorithm is based on divide and conquer: we explore a child node only if it is hit by the volume bounded by the hourglass. Note that this algorithm can be applied to arbitrary volumes, if they can give a *hit-cube* predicate. We thus traverse the octree of edges to get pairs of edges.

We then apply the algorithm detailed in 3.2 to compute the actual ESL(s) candidate(s) through the four edges. We then perform validation.

pseudo-ESLs enumeration

As stated in 5.2, some pseudo-ESLs have to be computed to give a consistent connectivity and structure to sub-swaths. These lines originate from configurations illustrated in figure 5.9. They contribute to inner discontinuities, but might still be computed if wanted. They are enumerated the following way: for each vertex of the scene, let $\pi_1 \dots \pi_n$ the planes supporting its surrounding faces. For each π_i hitting an edge of the source, build a pseudo-ESL from the intersection point of π_i and the source edge, to the vertex.

Validation is made the same way as regular ESLs, the only difference is that the generators of such a pseudo-ESL are not *enough*.

It is important to note that These pseudo-ESL were presented by Durands PhD thesis in [Dur99], as ESLs with face generators (more precisely:F_v).

Swath enumeration

In [DDP97], swath were not enumerated since the catalog provided connectivity. In our approach, we have to compute them separately, since they are not easily defined for degenerated ESLs.

The computed swaths are of the following kind:

- $V_sE\ E_sV$
- E_sEE

V_sE and E_sV swaths are computed exactly the same way as for point light source. That is, the swath is planar, has an apex at a vertex (on the source or not).

For E_sEE, we use the hourglass to identify potential swath, and then apply the swath validation algorithm. Indeed each edge hitting the hourglass might give a swath, since a set of lines run through the part of the edge which is inside the hourglass.

6.5 Meshing

We presented in the previous sections algorithms to compute shadow boundaries, and other discontinuities of the radiance function. In this section, we present techniques to subdivide the input geometry into elements on which the radiance function is smooth, up to a certain degree (say C^2 for example). In order to achieve this, we have to subdivide mesh elements along the discontinuities computed by the previous algorithms, that is along visibility events.

The input of the algorithm is:

- the input mesh
- the set of radiance discontinuities

The output of the algorithm is a subdivided mesh.

Constrained Delaunay Triangulation Approach

We suppose that blockers are faces (not clusters as proposed for optimisation).

Shadow boundaries, and by extend radiance function discontinuities, are given by the intersection of swaths and blockers (faces). These intersections are segments on the faces, which can be seen as constraints. We thus compute such intersections and use a constrained Delaunay triangulation (CDT) algorithm to subdivide the mesh. The constraint edges will define radiance function discontinuities, and on each sub-face, the radiance function will be smooth (up to a certain degree). The algorithm is thus made of two steps:

1. compute swath-face intersections
2. call a CDT algorithm

The second step is not detailed in this section. The reader should refer to triangulation literature for further information.

The first step is made in the same idea as intersection pre-computations.

Remember that along a given sub-swath, the visibility is constant, and especially, the receiver is the same. Boundaries of the sub-swath, given by ESLs, will intersect the receiver on boundaries of the constraint. We thus need to compute, for each sub-swath, the intersection of its boundary ESLs and the blocker of the sub-swath. The result gives the boundaries of the constraint on the receiver, and thus the constraint itself.

Note that the previous technique is only valid for planar swath, since the intersection of a planar swath with a face is a line segment. For quadric swaths, the intersection is a part of a conic. We approximate such a curved segment by sampling. Sampling of this curve can be done *before* projection, that is, the sub-swath is sampled along its reference edge (we take a point on the reference edge, and compute the line running through this point and the two edges), and the resulting line is intersected with the receiver. We thus approximate quadric constraints by a chain of line segment constraints.

The main drawback of this technique is robustness issues. Indeed, such algorithms suffer from robustness issues especially when constraints intersect, leading to computation of additional vertices which need proper placement in the structure which is sometimes difficult to handle.

Chapter 7

Implementation

7.1 Acceleration Structure

The algorithms presented here are oriented around lines. We compute the intersections of lines with objects such as spheres (for fat vertices), cylinders (for fat edges), and faces. These intersection predicates and computations are well known in graphics. A very popular algorithm has motivated research in this direction: ray-tracing.

Since our queries on the scene are very similar to the ones of ray-tracing, we will use the same data structures and acceleration techniques to improve our performances.

This section is separated into four parts: an introduction to the problem of casting a ray in a scene, a brief presentation on some data structures available (mainly the three fundamental types), our choice and their implementation.

Casting a ray

The problem of casting a ray in a scene is easily stated: given a scene of geometric objects, which one is the first hit by a ray, and where.

The first naive technique was to test for intersection against all the objects of the scene, and to get the closest (which is a linear complexity algorithm - in terms of input size). This complexity leads to a drastic loss of performances, as the scene complexity increases.

Glassner [Gla84] presented a technique to put objects into an octree: a tree of spacial cells, which at each node splits the node into eight sub-nodes if necessary. The tree is stored in a smart manner with a number related to its position in the space, avoiding confusion between a cell and its children.

The main benefit of the octree is that it has few empty cells, since a cell is split only if it holds too many objects. The main drawback is that some objects may appear twice in the traversal of the structure.

Goldsmith et al. [GS87] presented a heuristic for the optimal hierarchy of bounding box computations. This approach consists in putting together objects close to each-other considering them as a single bigger object.

Fujimoto et al. [AF86] presented a complete ray-tracing system with two types of acceleration structures: one with octrees, as for [Gla84], and the other with a grid (uniform space partitionning). They proposed a very efficient algorithm to traverse these data structures which was inspired from the line drawing algorithm. Amanatides and Woo [AW87] improved this algorithm to avoid divisions and other costy operations.

More complex and elaborated structures have been presented later, such as HUG, the Hierarchy of Uniform Grids by Cazals et al. [CDP95]. The work on the best structure still goes on, and the discussions on this topic are veray animated on ray-tracing news...

Existing Structures

We present here briefly the main structures used in ray-tracing and other space-subdivision oriented algorithms.

Octree The first structure we present here is the octree. The idea is very simple: given a box (axis-aligned), if it contains too many objects, we split it into eight non-overlapping boxes of half length. This splitting algorithm is recursive as well as the insertion algorithm and naive traversal with rays. The main benefit of this structure is its dynamic aspect. Indeed, as a hierarchic object, the structure can be localy modified easily. Its implementation is also easy in its naive approach.

Grid The second structure we present is the grid, or more precisely, the uniform grid. This structure is also simple: we subdivide the axis aligned bounding box of the scene along each axis. This subdivision depends on the number of objects in the grid. The main benefit of the grid is that a very fast traversal algorithm is known. The main drawback is that most cells are empty, and storage space is thus lost.

Hierarchy of Bounding Boxes This structure is an arbitrary hierarchy of potentially overlapping bounding boxes. This structure is relatively free in terms of implementation details and construction result, but the optimal hierarchy is not trivial to obtain. Its hierarchical aspect makes it dynamic in the same way the octree is.

Amongst the whole set of possible combination of structures, hybrids of grids, octrees, and hierarchies, we chose two types: the grid, with a particular implementation, and the tri-grid, which is impired by the octree and has 27 sub-cells.

Grids

In this section, we present the two structures we used: one is flat and the other is hierarchical. The first one is a simple grid, that is, given the bounding box of the scene (axis-aligned), we subdivide it in cubic-shaped cells with a fixed number of cells along each axis. The second one is a recursive grid with a fixed number of cells for each node split of the underlying hierarchy, which is 27 subcells ($27 = 3^3$, reason why we call it the tri-grid). We chose to use a fixed number mainly for performance issues, but we still wanted a finer refinement at each step than the case of the octree.

We present in this section the structures and the algorithms. For each of them, we consider the concept of a bounder: the bounder is a tool which can return the bounding box of an object, and test intersection between a given axis-aligned box and an object. The possibility to give the structure queries on the objects only through this bounder makes our approach general to any kind of geometry.

Simple Grid

The simple grid is static in our case, that is we do not add any cell (or insert elements in an empty cell). For a dynamic structure, we preferably use the trigrid. We give as input of the simple grid construction the set of objects, as well as a bounder. We then get the number of elements of the input, which interact with the grid. Given this number, we choose a number of cells.

The choice of number of cells has to be reasonably high so that the acceleration structure is effective, but not too high for memory consumption concerns. We classify the objects into 3 subclasses: BIG (more than a million elements), MEDIUM (between a thousand and a million), TINY (less than a thousant elements). Let n be the number of objects, we compute the number of significant bits k for our grid in the following way:

- BIG : $k = 2.17 * (ln(n) \quad sqrt(ln(n)))$
- SMALL : $k = 2.17 * (ln(n) \quad ln(ln(n)))$
- TINY : $k = 2.17 * ln(n)$

This empirical approach gives us a sufficient number of cells, keeping a reasonable grid size with respect to the available central memory.

The subdivision of the grid is done as follows : the number of cells along each axis is a power of two. The sum of these powers of two for each axis is k. We thus allocate k bits with a specific

distribution along axes. In order to have cubic-shaped cells, we allocate bits this way: let l_x, l_y and l_z the size of the bounding box of the grid along each axis. As long as we have bits left, we increment the number of bits allocated to a direction for the axis which has the greatest value of l and divide this size value by two. We finally slice the bounding box of the grid along each axis for the given number of cells by axis.

We then insert the elements into each cell which hits the element (we use the bounding box of the elements for faster insertion).

The Tri-Grid

The tri-grid is our dynamic structure. It is constructed progressively during insertion or deletion of objects from the structure. The refinement criterion is simple: if we reach a given number of elements in a cell (parameter of the tri-grid), we split the cell into 27 subcells.

The tri-grid contains the root of the hierarchy, which is a cell of the size of the grid. Each cell has a list of elements, or a list of children.

Implementation

Elements

Acceleration structures work with rays and elements: a ray is cast in the structure and returns the elements it successively hit. For generality reasons, elements derive from an abstract empty class, and is simply a typed pointer. The queries of the acceleration structure to the elements are done via a bounder. The bounder has two functions: returning the bounding box of an object and answering the boolean query of intersection between an object and an axis-aligned box. Rays are given as an origin and a direction in the world space.

Simple Grid

The simple grid has many empty cells. It would be a waste to store them all with the flag *empty*. We thus use a hollow-array for cell storage. A hollow array is a structure which only stores non-empty cells, a quick query is achieved to test if a cell (index in the hollow array), is stored (ie non empty).

A cell is completely identified by a pointer to an axis-aligned box (of the grid), and a 32 bit number. Indeed, we do not allow grids of more than 4G cells, so the number of the cell is a 32 bits number. As we split the grid along the axis by powers of two, the quantized position of a cell has a coordinate value from 0 to $2^k \quad 1$, where k is the number of bits allocated for this coordinate. We can naturally encode a cell number with the value of its quantized positions.

For the traversal algorithm, we used the algorithm defined by [AW87] for a grid traversal with few operations. Going from one cell to another is simply an increment or decrement of the cell's

quantized coordinate, and more efficiently, an addition or substraction on the cell index. Overflow and underflow can be tested with a mask on the index number.

The structure has not been benchmarked, only a result has been established: for a given model (bunny 69k polygons), we cast an average of 350,000 rays per second in the scene.

Tri-Grid

The tri-grid, on the other hand has fewer cells, and at most 26 empty cells per node. We did not use any particular way to store cells, or allocate new cells when a split occurs. This part could clearly be optimized.

For each cell, we associate an index: we quantize uniformly by 729 (3^6) each coordinate of the minimal corner of its bounding box. We then encode each position in ternary representation. The terns of each coordinates are then concatenated to form a value between 0 and $27d$. This value is encoded on 5 bits. The position of the cell holds in 30 bits, with highest weight five bits for highest tern of the coordinates, and so on. We use tables to avoid multiplications and divisions. In order to differenciate the root cell from a child cell at minimal position $(0, 0, 0)$, we use the following technique: for a given level of the hierarchy, all bits are not significant, only the $3 * n$th first are (n being the depth of the tree, 0 for root). We thus end our cell number with ones. We are not confused with a subcell since $11111b$ is not between 0 and 27 and is thus not a cell number. (The same kind of technique has been used by Glassner in [Gla84] for octree cells numbered from 1 to 8 instead of 0 to 7.) For example, the root node is $0xFFFFFFFF$, and the children are numbered from $0x07FFFFFF$ to $0xDFFFFFFF$. The children of cell $0x07FFFFFF$ are numbered from $0x003FFFFF$ to $0x077FFFFF$.

Any position for a query is uniformly quantized by 729 possible positions for each coordinate (which would result in a grid of size 387 million cells, which is enough for our needs). The position is then encoded the same way cell minima are (without any ones filling). To access the node, we get the highest 5 bits of the encoded position - noted index -, and start from the root node. We shift the encoded position of 5 bits, and step to the child (if exists) of index. We recursively traverse the cell tree until we are in a leaf node, which is our result.

The traversal algorithm is based on this idea: we compute the quantized position of the entry point in the next cell, and get the cell from the previous algorithm.

The structure has not been benchmarked, only a result has been established: for a given model (bunny 69k polygons), we casted an average of 175,000 rays per second in the scene.

Mesh Data Structures

Along all this description of algorithms, we made the assumption that we had immediate access to edge description and connectivity. We describe here briefly the main types of mesh data structures, as well as ours. Each structure has its benefit and its drawbacks.

We will present the half-edge, the quad-edge, the split-edge and the corner data-structures. We then describe our choice on the data structure which has multi-layer information.

Well-Known structures

We present in this section the existing mesh data-structures. This is an exhaustive list to the extend of our knowledge.

Half-Edge

The half-edge data structure is, as its name states, a structure oriented around edges. Each edge is split into two oriented half-edges, one in each direction. For each of these half-edge, we store the following information:

- the vertex it points to
- the sibling half-edge
- the next edge in a face loop

Figure 7.1 illustrates this structure.
A vertex is represented as its coordinates in a table.
A face is represented by one of its half-edge.
Lets consider an example mesh: the Stanford bunny. The number of faces is $69,473$, the number of vertices is $34,835$, and the number of half-edges is $208,620$. The memory cost of this structure is thus: $208,620*3+34,835*3+69,473 = 799,838$, in terms of pointers, which is $3,199,352$ bytes.

The complexity of various access and loop algorithms is constant for the neighboring connectivity information, except for the face access given an edge. This access is either logarithmic in terms of the mesh size (find an edge of the face-half-edge loop which is actually the entry for a face), or we have to change the structure and add a face pointer in it (making the structure reach 4Mb).

This structure would be very efficient in terms of access time (with the extension), and its implementation could fulfill most of our needs. But since we want to be able to access all kinds of geometry, whether it be solid or not, badly linked (three faces on the same edge), we preferred to use another kind of data-structure for our meshes.

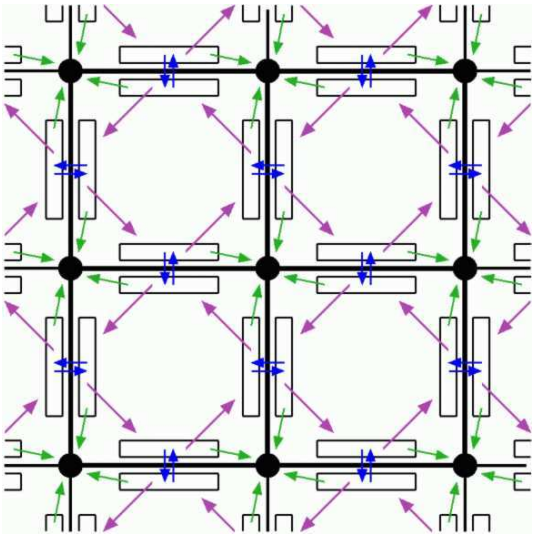


Figure 7.1: The Half-Edge data structure from http://www.graphics.lcs.mid.edu/ legakis/6838/598/VIII.SplitEdge.and.Corner.html

Quad-Edge

The quad-edge data structure is oriented around edges, but in a different manner. Each quad-edge has the following information: two references to vertices, and two references to faces (connected). As illustrated in 7.2, we can see the structure in action. For example, the edge named g connects vertices 6 and 2 (drawn as circles on the top-right). It is also a boundary for the face F. A face can thus be represented by one of its edge and a side, and the vertex in an array of coordinate values.

For the same bunny example, the cost is: $208,620 * 4 + 34,835 * 3 + 69,473 = 1,008,458$, in terms of pointers, which is about 4Mb. The same remark for faces apply in this case, which would lead us to 4.8Mb.

Split-Edge

The split-edge is sometimes presented as the dual of the half-edge. Each edge is also split into two half-edges, but in a different manner. The difference is in the next edge, which also points to the same vertex, making vertices loop algorithm as face loops of the half-edge and vice-versa. An illustration is given in figure 7.3.

The storage cost is exactly the same as the half-edge.

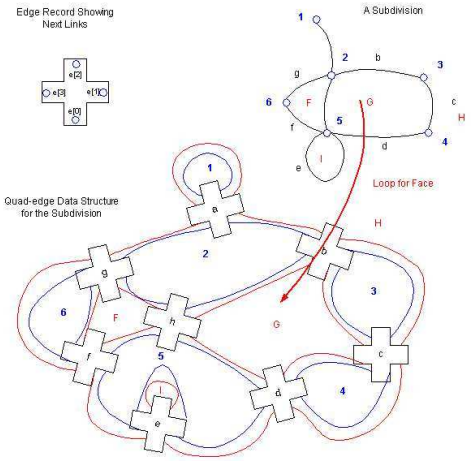


Figure 7.2: The Quad-Edge data structure from http://www.stanford.edu/ rakbas/solid/quadedge.hml

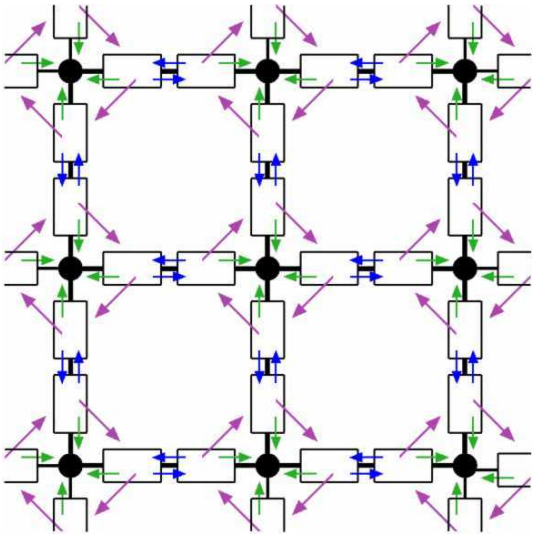


Figure 7.3: The Split-Edge data structure from http://www.graphics.lcs.mid.edu/ legakis/6838/598/VIII.SplitEdge.and.Corner.html

Corner

The corner data structure is oriented around corners of each face. Each corner has a pointer to the vertex it refers, and two pointers to neighboring edges. The structure is illustrated figure 7.4.

For example, given a corner, you can iterate to the next corner of the face, or switch to a connected face. The capabilities are the same as the half-edge structure.

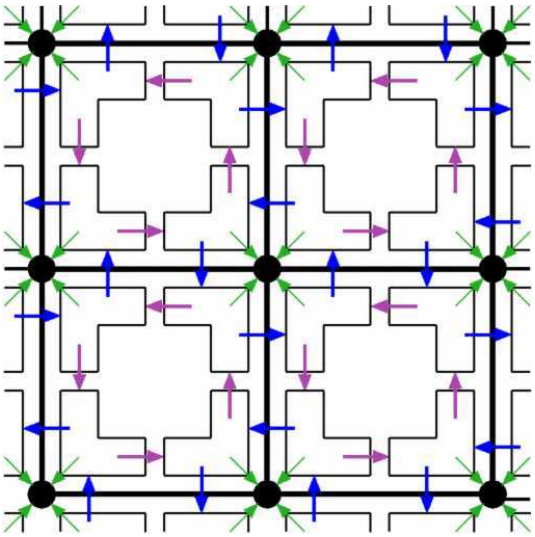


Figure 7.4: The Corner data structure http://www.graphics.lcs.mid.edu/ legakis/6838/598/VIII.SplitEdge.and.Corner.html

The main drawback of this approach is that the edge information is not directly available, but can still be encoded easily (the corner for which the next corner pointer connects to the connected edge). The same concept of half-edge is also present here. The same drawback for faces is also present.

The memory cost of this structure for the example of the bunny is: 3.2Mb

Our Structure : Multi layer connectivity information

We designed a structure with multi-layer connectivity information giving the ability to only store "permanently" minimal, but sufficient information, and to restore acceleration structures (for connectivity access) when needed. This structure is made of three layers:

1. The first layer is the fundamental layer. It stores all the vertices in an array of coordinates and the faces with a double entry table. (Exactly the same way VRML or IV does for indexed face sets). This first layer construction is cost-less since the meshes are stored this way in most 3d Graphics file formats.

2. The second layer creates edges. With this layer, we have access to the connectivity between vertices through the same concepts as half edges. Besides, the half edges are stored implicitly (the vertex which is pointer to by the edge).

3. The third layer creates an additional acceleration structure to grant access to the face connectivity. With this structure, vertices can have access to neighboring faces, and edges also.

The structure of the first layer is the following: given the number of vertices and the number of faces, we create two tables: one of floating point values for the coordinates (three times the number of vertices - meshes in 3D), and another for face indices, given entry-points for faces in another table (vertices of the faces). This structure is the regular way to store indexed face sets in modeling files.

The structure of the second layer is the following: we create a double-entry array for vertices connectivity (the same way we did for faces). The sub-array of elements for each vertex is the set of connected vertices. Each entry of the big table corresponds to an actual half edge, but the information is not immediately available. Besides, given this second table, we do not have access to face connectivity information. The main benefit of this approach is that we have a unique id for each half-edge: its index in the table.

The final layer is the face connectivity for vertices. It is stored exactly the same way edges are. Each entry in the first table gives access to the set of faces connected to each vertex. To get the faces connected to an edge, we need to get the face connected to the one vertex of the edge, which is also connected to the other. There are two faces, with different indices in the table, which helps us differentiate them. Note that besides this apparent complexity to retrieve the faces connected to an edge, it is still a local computation and is bound by the valency of the vertices.

The cost of this structure for the example of the bunny is (by layers):

1. vertices: $418,020$ bytes, faces : $1,111,572$ bytes.

2. index-table: $139,344$ bytes, data-table: $834,480$ bytes.

3. index-table: $139,344$ bytes, data-table: $833,676$ bytes.

The overall cost is $1,529,592 + 973,834 + 973,020 = 3,476,446$ bytes. This result is better than all the previous one if we consider the local-time access for face connectivity. The main benefit of this structure is of course its multi-layer aspect. Once computations are done, face connectivity might not be needed and the structure with edge connectivity would be less costly than previous ones.

Basic Geometric Algorithms

We present here two basic algorithms we extensively use: the intersection between a ray and a sphere, and the intersection between a ray and a cylinder. Amongst all tutorials web pages and other publications, we viewed a lot of various algorithms with for each of them benefits and drawbacks. We present here two algorithms which satisfy our needs: robustness and precision.

A ray is defined by an origin O, and a direction $\vec{u}$. The points of the ray are given by $M(\lambda) = O + \lambda\vec{u}$, with $\lambda \geq 0$.

A sphere is defined by its center C and its radius r.

A cylinder is defined by its two extremal points (centers of extremal discs) A, B, and its radius r.

We need the precisions to be much greater (so error much smaller) than the epsilon we mean to use. A precision of 10^{-5} will certainly be not satisfactory !!

Intersection of a ray and a sphere

The intersection points are given by the points on the ray with parameters, the solutions of the equation $\vec{CM} \cdot \vec{CM} = r^2$, that is with substitution of the M point by its expression on the ray:

$$\vec{u} \cdot \vec{u} + 2\vec{u}\vec{CO} + \vec{CO} \cdot \vec{CO} \quad r^2 = 0$$

This second degree equation is easy to solve. Besides, the discriminant of this polynomial is not robustly computed, especially when O is far from C. We thus use the following technique illustrated figure 7.5.

We compute the closest points in homogeneous coordinates to avoid divisions:

$$I = O + \frac{\vec{OC} \cdot \vec{u}}{\vec{u} \cdot \vec{u}}\vec{u}$$

Then we compute the distance between this point and the center of the sphere:

$$d^2 = \cdot CI \cdot \vec{CI}$$

in an homogeneous manner (that is we multiply C by $\vec{u} \cdot \vec{u}$, and obtain $d^2 * \vec{u} \cdot \vec{u}$). We compare to the square radius to do our hit test (yet, only multiplies and additions, which are almost free on the ix86 family). Finally, we get the two intersection parameters by increasing and decreasing this value by the square root of this square distance (once-again in an homogeneous manner). We thus avoid divisions.

The performances are the following: on a PIV Xeon 2GHz, an intersection requires on average 96 νs.

The precision is of order 10^{-9} on the distance between the sphere and the ray (for computations using double precision arithmetic).

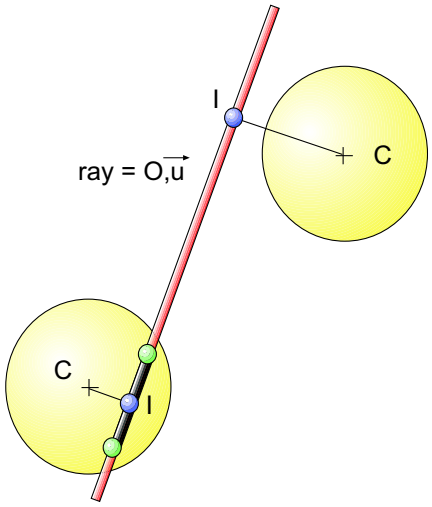


Figure 7.5: The ray intersect sphere algorithm

Intersection of a ray and a cylinder

This algorithm is subdivided into two parts: the intersection between a ray and an infinite cylinder, and the clamp of the result between the two planes. The clamp is defined by the intersection between the ray and the planes orthogonal to the axis of the cylinder, et the vertices positions. This clamp can be made in homogeneous coordinates without any divisions.

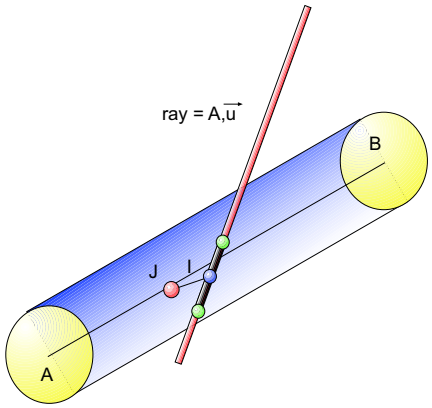


Figure 7.6: The ray intersect cylinder algorithm

The intersection is illustrated figure 7.6. In order to compute the distance between the ray and the infinite cylinder, we compute the distance between a point on the ray (say $V = O + \mu\vec{u}$), and the nearest point on the cylinder:

$$I = A + \frac{(\vec{OA} + \mu\vec{u}) \cdot \vec{AB}}{\vec{AB} \cdot \vec{AB}} \vec{AB}$$

The square distance between the two points is given by: $\vec{VI} \cdot \vec{VI}$, with

$$\vec{VI} = \{\vec{OA} \quad \frac{\vec{OA} \cdot \vec{AB}}{\vec{AB} \cdot \vec{AB}} \vec{AB}\} + \mu\{\vec{u} \quad \frac{\vec{u} \cdot \vec{AB}}{\vec{AB} \cdot \vec{AB}} \vec{AB}\}$$

Which we can rewrite using the double cross product expression:

$$\vec{VI} = \vec{o}_{-} + \mu\vec{u}_{-}\vec{o}_{-} = \vec{AB} \times \vec{OA} \times \frac{\vec{AB}}{AB^2}\vec{u}_{-} = \vec{AB} \times \vec{u} \times \frac{\vec{AB}}{AB^2}$$

The square distance d^2 is finally given by:

$$AB^2 d^2 = \mu^2(\vec{u} \times \vec{AB})^2 + 2\mu(\vec{u} \times \vec{AB}) \cdot (\vec{OA} \times \vec{AB}) + (\vec{OA} \times \vec{AB})^2$$

We can then use a trinomial resolution to extract the two roots and return the values of μ for which the distance is equal to the radius. These operations do not imply any subdivisions and clamping the results can be done with homogeneous values.

The performances are the following: on a PIV Xeon 2GHz, an intersection requires on average 256 νs.

The precision is of order 10^{-7} on the distance between the cylinder and the ray (for computations using double precision arithmetic).

Chapter 8

Conclusion

In this thesis, we have presented a novel framework for analytic visibility based on epsilons. This epsilon visibility provides algorithms, techniques and other tools to perform robust visibility computations in large 3D environments. We provided several custom solutions to problems specific to our needs such as robust and efficient intersection of rays and cylinders, or static and very fast acceleration structures for ray tracing.

This project is not fully finished, and we expect new results soon. The main algorithms related to strict visibility computations will soon be used in a global illumination algorithm to bridge the gap between geometry and lighting. Extensions of this work are also expected in other fields such as occlusion culling.

The most promising part of this work is its potential regarding hierarchical visibility. Indeed, since visibility algorithms are of complexity at least quadratic, increasing geometric complexity is not yet possible. An extension to our epsilon approach using a very large value compared to the size of the objects could be a first approach to hierarchical visibility.

Appendix A

Line Space and Plücker Coordinates

This section is a short introduction to line space and Pl"ucker coordinates [Plü65].

We present here geometrical tools useful for lines handling, especially in an Euclidean 3D space (world space, within which the scene is described). These tools mainly originate from Euclidean geometry, and linear algebra. By the way, for Plücker coordinates, some notions on varieties might be useful even though not needed.

In this part, we make the assumption that between two points lies one and only one unoriented line, or two oriented lines.

We introduce line space issues with 2D lines representation problems, then we introduce Plücker coordinates with some algorithms.

Introduction

In 2D space, lines can be represented in different manners (this list is not exhaustive):

- by its intersection with the Oy axis and its angle with the Ox axis: (y_0, θ)
- by an equation of the kind: $ax + by + c = 0$ (no orientation)
- by a point and a direction vector: $M_0, \vec{u}$

Note that each representation has a different number of parameters: two for the first one, three for the second one and four for the last one.

The first representation has a continuity problem since if the line is parallel to the Oy axis, such parametrisation is not possible. In order to avoid specific treatment, we leave this line representation for another.

In the two following representations, we can note redundancy. In the second one, the parametrisation is done in a projective space, that is if we multiply each parameter by a non null constant, the line represented is the same. In the last one, the point may be chosen anywhere on the line, and the

direction vector can be scaled. Besides, The second parametrisation does not allow orientation of lines, since it is a set definition (points on the line have coordinates which satisfy the given equation).

The other benefits of the last representation is that its definition is also valid in 3D, whereas the second needs two equations to be described, raising the number of parameters to 8.

Plücker coordinates are a parametrisation of 3D lines close to the last one. We will present in the following section, the Plücker coordinates, and some tools to use them.

Plücker Coordinates

Parametrisation

Let P and Q be two points in the 3D space, with coordinates (x_p, y_p, z_p) for P and (x_q, y_q, z_q) for Q. The six Plücker coordinates are the six determinants computed from two columns of the following matrix:

$$\begin{pmatrix} x_p & y_p & z_p & 1 \\ x_q & y_q & z_q & 1 \end{pmatrix}$$

That is:

$$(PQ) = \begin{pmatrix} \pi_l 0 \\ \pi_l 1 \\ \pi_l 2 \\ \pi_l 3 \\ \pi_l 4 \\ \pi_l 5 \end{pmatrix} = \begin{pmatrix} x_p y_q & y_p x_q \\ x_p z_q & z_p x_q \\ x_p & x_q \\ y_p z_q & z_q y_p \\ z_p & z_q \\ y_q & y_p \end{pmatrix}$$

This parametrisation is strictly equivalent to the following one (permutations and sign change):

$$(PQ) = (\vec{u}, \vec{v}) = \left(\overrightarrow{PQ}, \overrightarrow{OP} \times \overrightarrow{OQ}\right)$$

From now one, we will keep the last representation of the line which is more intuitive, and easy to write / handle. Note that the $\vec{u}$ vector is in fact the direction vector of the line. Also note that with this construction $\vec{u} \cdot \vec{v} = 0$. See figure A.1 for illustration.

Redundancies

This six parameters representation is redundant in two ways: the first is that this representation is invariant with a strictly positive scaling. We are in a projective space. We call the unit cylinder, the five dimension algebraic variety defined by $\vec{u} \cdot \vec{u} = 1$, that is the set of lines with unit direction vector. We call this cylinder the normalisation variety (or cylinder), despite the non unit value of the six dimension vector's norm. The second one has been raised in the construction step. Indeed all sextuple of parameters do not represent lines through this parametrisation since we shall always have $\vec{u} \cdot \vec{v} = 0$. Any line which does not satisfy this equation is said to be unreal or imaginary, in

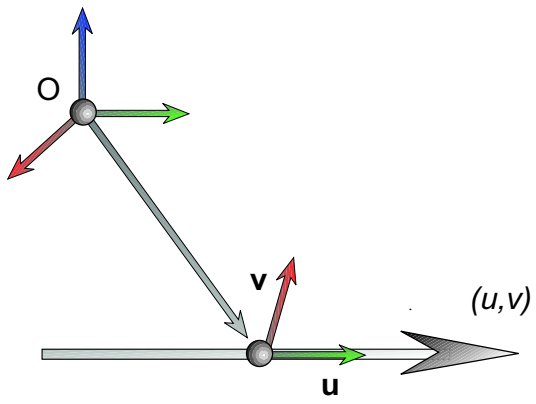


Figure A.1: The Plücker line

contrast with real lines which satisfy this equation. The algebraic variety associated with the last equation is called the reality variety (or cone).

The intersection of the two varieties define a four dimension variety, which we consider to be our line space. From now on, we will only consider lines as parametrisations of this space. Any other element of the 6D space will be temporary, in the context of computations.

The Plücker Bilinear Form

Let $\delta_1 = (\vec{u}_1, \vec{v}_1)$ and $\delta_2 = (\vec{u}_2, \vec{v}_2)$ be two lines. The Plücker bilinear form, noted $\odot$, of these two lines is given by:

$$\delta_1 \odot \delta_2 = \vec{u}_1 \cdot \vec{v}_2 + \vec{u}_2 \cdot \vec{v}_1$$

Note that for any (real) line δ, $\delta \odot \delta = 0$.

This bilinear form is symmetric and gives the orientation of a line with respect to the other, see figure A.2 for illustration. Note that this bilinear form is not a dot product since its eigen values are 1 and 1, triple times.

Two lines have a null Plücker form if they are parallel or they intersect. Indeed, in these cases the orientation cannot be defined.

Intrinsic Line Parametrisation

One of the main benefits of such a representation is that it has an intrinsic line parametrisation, that is given these coordinates, a point of the line is uniquely defined by a single real value. The origin of this intrinsic parametrisation is given by the following equation (we suppose once again the line to have a unit vector):

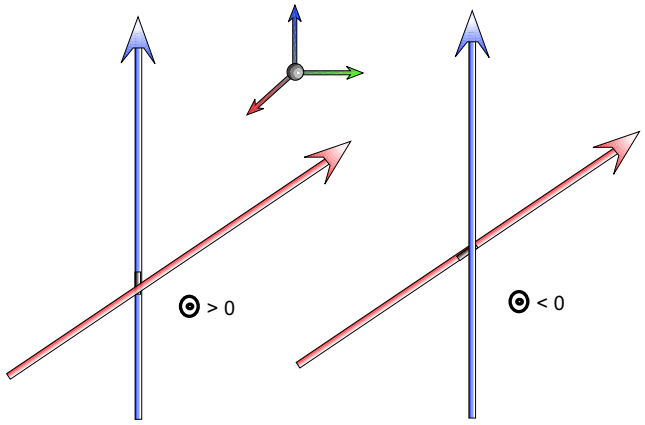


Figure A.2: Illustration of the Plücker form

$$\Omega = \vec{u} \times \vec{v}$$

The points on the line are then given by the following equation, for λ an arbitrary real value:

$$M_\lambda = \Omega + \lambda \vec{u}$$

Also, an immediate projection algorithm is available: the parameter of the nearest point on the line from point P is given by:

$$\lambda = \overrightarrow{OP} \cdot \vec{u}$$

Some Algorithms using Plücker coordinates

Intersection With a Plane

Let π be a plane of normal $\vec{n}$ and distance to the origin d. The intrinsic parameter of the point is given by:

$$\begin{array}{rcl} \lambda & = & \frac{d \quad [\vec{u},\vec{v},\vec{n}]}{\vec{u}\cdot\vec{n}} \\ [\vec{u},\vec{v},\vec{n}] & = & (\vec{u} \times \vec{v}) \cdot \vec{n} \end{array}$$

Line From Two Planes

Let π be a plane of normal $\vec{e}$ and distance to the origin e, and ρ with $\vec{f}$, and f. The Plücker parametrisation of the line intersection of the two planes is given by:

$$\vec{u} = \vec{e} \times \vec{f}$$
$$\vec{v} = \frac{f\vec{e} \quad e\vec{f}}{1 \quad (\vec{f}\cdot\vec{e})^2}$$

Proof: Is Ω on the planes ?

$$\begin{array}{rcl} \Omega & = & \vec{u} \times \vec{v} \\ \Omega & = & (\vec{e} \times \vec{f}) \times \frac{f\vec{e} \quad e\vec{f}}{1 \quad (\vec{f}\cdot\vec{e})^2} \\ \left[1 \quad (\vec{f}\cdot\vec{e})^2\right]\Omega & = & f\left[\vec{f} \quad (\vec{f}\cdot\vec{e})\vec{e}\right] \quad e\left[(\vec{e}\cdot\vec{f})\vec{f} \quad \vec{e}\right] \\ \left[1 \quad (\vec{f}\cdot\vec{e})^2\right]\Omega\cdot\vec{e} & = & f(\vec{f}\cdot\vec{e} \quad \vec{f}\cdot\vec{e}) \quad e\left[(\vec{e}\cdot\vec{f})^2 \quad 1\right] \\ \left[1 \quad (\vec{f}\cdot\vec{e})^2\right]\Omega\cdot\vec{f} & = & f\left[1 \quad (\vec{f}\cdot\vec{e})^2\right] \quad e(\vec{e}\cdot\vec{f} \quad \vec{e}\cdot\vec{f}) \end{array}$$

Parameters of Nearest Points

Let $\delta_1 = (\vec{u}_1, \vec{v}_1)$ and $\delta_2 = (\vec{u}_2, \vec{v}_2)$ be two lines. The aim of this algorithm is to compute the parameters of the nearest point on a line of the other. That is the parameter of I_1, point of δ_1 nearest from δ_2, (and vice versa).

We first compute the following elements:

$$\begin{array}{rcl} s & = & \vec{u_1}\cdot\vec{u_2} \\ m_1 & = & [\vec{u}_1, \vec{v}_1, \vec{u}_2] \\ m_2 & = & [\vec{u}_2, \vec{v}_2, \vec{u}_1] \end{array}$$

The nearest points are so that $I_1\overrightarrow{I_2}$ is orthogonal to both $\vec{u}_1$ and $\vec{u}_2$. We note $I_1 = \Omega_1 + \lambda\vec{u}_1$ and $I_2 = \Omega_2 + \mu\vec{u}_2$. And we right the vector equation, giving:

$$\begin{array}{rcrcl} \lambda & & \mu s & = & m_2 \\ \lambda s & + & \mu & = & m_1 \end{array}$$

We then invert the matrix, and obtain the result:

$$\begin{array}{rcl} \lambda & = & \frac{m_2 + s m_1}{1+s^2} \\ \mu & = & \frac{m_1 + s m_2}{1+s^2} \end{array}$$

Note that this algorithm is very useful to find the nearest point on an edge on a given ray.

Bibliography

[AF86] Kansei Iwata Akira Fujimoto, Takayuki Tanaka, *Arts : Accelerated ray-tracing system*, IEEE Computer Graphics and Applications (1986), 16–26.

[ARHM00] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll, *Efficient image-based methods for rendering soft shadows*, ACM SIGGRAPH 2000, Annual Conference Series, July 2000, pp. 375–384.

[Arv94] J. Arvo, *The irradiance Jacobian for partially occluded polyhedral sources*, ACM SIGGRAPH '94, 1994, pp. 343–350.

[AW87] John Amanatides and Andrew Woo, *A fast voxel traversal algorithm for ray tracing*, Eurographics '87, Elsevier Science Publishers, Amsterdam, North-Holland, 1987, pp. 3–10.

[BDT99] K. Bala, J. Dorsey, and S. Teller, *Radiance interpolants for accelerated bounded-error ray tracing*, ACM Transactions on Graphics **18** (1999), no. 3, 213–256.

[Cat74] Edwin E. Catmull, *A subdivision algorithm for computer display of curved surfaces*, Ph.D. thesis, Dept. of CS, U. of Utah, December 1974.

[CDP95] Frédéric Cazals, George Drettakis, and Claude Puech, *Filtering, clustering and hierarchy construction: a new solution for ray tracing very complex environments*, Eurographics '95, 1995.

[CF90] A. T. Campbell, III and D. S. Fussell, *Adaptive mesh generation for global diffuse illumination*, Computer Graphics (Proc. SIGGRAPH '90) **24** (1990), 155–164.

[CF92] N. Chin and S. Feiner, *Fast object-precision shadow generation for areal light sources using BSP trees*, Computer Graphics (1992 Symposium on Interactive 3D Graphics), vol. 25, March 1992, pp. 21–30.

[CLO98] David A. Cox, John B. Little, and Donal O'Shea, *Ideals, varieties, and algorithms*, Springer, 1998.

[Cro77] F. C. Crow, *Shadow algorithms for computer graphics*, Computer Graphics (Proc. SIGGRAPH 77) **11** (1977), no. 2, 242–248.

[DD02] Florent Duguet and George Drettakis, *Robust epsilon visibility*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM Press, 2002, pp. 567–575.

[DDP96] Frédo Durand, George Drettakis, and Claude Puech, *The 3d visibility complex, a new approach to the problems of accurate visibility*, Proceedings of 7th Eurographics Workshop on Rendering in Porto, Portugal (Rendering Techniques '96) (Xavier Pueyo and Peter Schröder, eds.), Springer Verlag, June 1996, pp. 245–256.

[DDP97] _____, *The visibility skeleton: A powerful and multi-purpose global visibility tool*, ACM SIGGRAPH 97, August 1997, http://w3imagis.imag.fr/Membres/Fredo.Durand/PUBLI/siggraph97/index.htm.

[DDP02] Frédo Durand, George Drettakis, and Claude Puech, *The 3d visibility complex*, ACM Transactions on Graphics **21,2** (2002).

[DF94] George Drettakis and Eugene Fiume, *A fast shadow algorithm for area light sources using backprojection*, Proceedings of SIGGRAPH '94 (Andrew Glassner, ed.), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, 1994, pp. 223–230.

[DHH01] Olivier Devillers and Olaf Hall-Holt, *Predicates and constructions for visibility problems*, manuscrit, 2001.

[Duf92] T. Duff, *Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry*, Computer Graphics (Proc. SIGGRAPH'92) **26** (1992), no. 2, 131–138.

[Dur99] Frédo Durand, *3d visibility: analytical study and applications*, Ph.D. thesis, Université Joseph Fourier, Grenoble I, July 1999, http://www-imagis.imag.fr.

[EK] Cass Everitt and Mark J. Kilgard, *Practiral and robust stenciled shadow volumes for hardware accelerated rendering*, `http://developer.nvidia.com`.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller, *Hierarchical z-buffer visibility*, ACM SIGGRAPH '93, 1993.

[Gla84] Andrew S. Glassner, *Space subdivision for fast ray-tracing*, IEEE Computer Graphics and Applications (1984), 15–22.

[GM90] Z. Gigus and J. Malik, *Computing the aspect graph for the line drawings of polyhedral objects*, IEEE Trans. Pattern Analysis and Machine Intelligence **12** (1990), no. 2.

[GS87] Jeffrey Goldsmith and John Salmon, *Automatic creation of object hierarchies for ray-tracing*, IEEE Computer Graphics and Applications (1987), 14–20.

[Hai93] E. A. Haines, *Shaft culling for efficient ray-traced radiosity*, Photorealistic Rendering in Comp. Graphics, Springer Verlag, 1993, Proc. 2nd EG Workshop on Rendering (Barcelona, 1991), pp. 122–138.

[Hec91] Paul S. Heckbert, *Simulating global illumination using adaptive meshing*, Ph.D. thesis, CS Division, UC Berkeley, June 1991, Tech. Report UCB/CSD 91/636.

[Hec92] _____, *Discontinuity meshing for radiosity*, Eurographics Rendering Workshop 1992, Eurographics, May 1992, pp. 203–216.

[JW89] David Jevans and Brian Wyvill, *Adaptive voxel subdivision for ray tracing*, Proceedings Graphic's Interface '89, Canadian Information Processing Society, 1989, pp. 164–172.

[Kaj86] James T. Kajiya, *The rendering equation*, Computer Graphics (SIGGRAPH '86 Proceedings) (David C. Evans and Russell J. Athay, eds.), Computer Graphics Proceedings, Annual Conference Series, vol. 20,4, ACM SIGGRAPH, ACM Press, Août 1986, pp. 143–150.

[KvD79] Jan J. Koenderink and Andrea J. van Doorn, *The internal representation of solid shape with respect to vision*, BioCyber **32** (1979), 211–216.

[LP00] L. Leblanc and P. Poulin, *Guaranteed occlusion and visibility in cluster hierarchical radiosity*, Proc. Eurographics Workshop on Rendering 2000, June 2000, pp. 89–100.

[LTG92] D. Lischinski, F. Tampieri, and D. P. Greenberg, *Discontinuity meshing for accurate radiosity*, IEEE CGA **12** (1992), no. 6, 25–39.

[nvi] nvidia, *webpage*, `http://developer.nvidia.com/view.asp?IO=cedec_stencil`.

[Plü65] Plücker, *On a new geometry of space*, Phil. Trans. Royal Soc. London, 1865.

[PV96] Michel Pocchiola and Gert Vegter, *The visibility complex*, International Journal of Computational Geometry and Applications **6** (1996), no. 3, 279–308.

[RDO79] Ramis, Deschamps, and Odoux, *Cours de mathématiques spéciales*, vol. 2, Masson, 1979.

[Riv95] Stéphane Rivière, *Topologically sweeping the visibility complex of polygonal scenes*, Proceedings of the eleventh annual symposium on Computational geometry, ACM Press, 1995, pp. 436–437.

[Riv97] _____, *Dynamic visibility in polygonal scenes with the visibility complex*, Proceedings of the thirteenth annual symposium on Computational geometry, ACM Press, 1997, pp. 421–423.

[SD02] Marc Stamminger and George Drettakis, *Perspective shadow maps*, Proceedings of ACM SIGGRAPH 2002 (John Hughes, ed.), Annual Conference Series, ACM Press/ ACM SIGGRAPH, July 2002.

[SG94] A. James Stewart and Sherif Ghali, *Fast computation of shadow boundaries using spatial coherence and backprojections*, Proceedings of SIGGRAPH '94 (Andrew Glassner, ed.), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, 1994, pp. 231–238.

[SGHS98] J. W. Shade, S. J. Gortler, L. He, and R. Szeliski, *Layered depth images*, Computer Graphics Proceedings, Jul 1998, Annual Conference Series, SIGGRAPH'98, pp. 231–242.

[SGS89] D. Salesin, L. Guibas, and J. Stolfi, *Epsilon geometry: Building robust algorithms from imprecise computations*, Annual Symposium on Computational Geometry, 1989, Saarbrucken, West Germany.

[SK98] A. James Stewart and Tasso Karkanis, *Computing the approximative visibility map, with applications to form factor and discontinuity meshing*, Eurographics Rendering Workshop 1998, Eurographics, 1998.

[Sny92] J. M. Snyder, *Interval analysis for computer graphics*, Computer Graphics (Proc. SIGGRAPH'92) **26** (1992), no. 2, 121–130.

[SR00] Michael M. Stark and Richard F. Riesenfeld, *Exact radiosity reconstruction and shadow computation using vertex tracing*, Proceedings of 11th Eurographics Workshop on Rendering, 2000.

[SR01] ———, *Reflected and transmitted irradiance from area sources using vertex tracing*, Proceedings of 12th Eurographics Workshop on Rendering, 2001.

[SS98] Cyril Soler and François Sillion, *Fast calculation of soft shadow textures using convolution*, Computer Graphics Proceedings, Jul 1998, Annual Conference Series, SIGGRAPH'98, pp. 321–332.

[Tel92a] Seth J. Teller, *Computing the antipenumbra of an area light source*, Proceedings of SIGGRAPH '92, Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, 1992, pp. 139–148.

[Tel92b] ———, *Visibility computation in densely occluded polyhedral environments*, Ph.D. thesis, University of California, Berkeley, 1992.

[TH94] Seth Teller and Pat Hanrahan, *Global visibility for illumination computations*, Proceedings of SIGGRAPH '94, Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, 1994, pp. 443–450.

[TS91] Seth J. Teller and Carlo H. Séquin, *Visibility preprocessing for interactive walkthrough*, ACM SIGGRAPH '91, july 1991, pp. 61–69.

[VG97] Eric Veach and Leonidas J. Guibas, *Metropolis light transport*, SIGGRAPH 1997 Proceedings, Annual Conference Series, Addison-Wesley, August 1997, pp. 65–76.

[WA77] K. Weiler and K. Atherton, *Hidden surface removal using polygon area sorting*, Computer Graphics (Proc. SIGGRAPH 77) **11** (1977), no. 2, 214–222.

[Wil78] Lance Williams, *Casting curved shadows on curved surfaces*, Proceedings of SIGGRAPH '78, ACM SIGGRAPH, August 1978, pp. 270–274.

[WPF90] Andrew Woo, Pierre Poulin, and Alain Fournier, *A survey of shadow algorithms*, IEEE Computer Graphics and Applications **10** (1990), no. 6, 13–32.

[ZMHI97] Hanson Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III, *Visibility culling using hierarchical occlusion maps*, ACM SIGGRAPH '97, 1997.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399